Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 5

Polymorphism

Polymorphism

- One of the most important principles of object-oriented programming

- Stands in the basis of many other object-oriented principles
  - Class vs. Object
  - Reference vs. Content
  - Encapsulation
  - Inheritance and interfaces
  - **Extensibility**
  - **Modularity**

Polymorphism
Definition

- From Greek – “many shapes”
  - A biological term in which an organism or species can have many different forms or stages

- In object oriented programming, polymorphism refers to the ability of an object to **take on many forms** (i.e., **types**)

- This ability is realized both when **extending** a **class** and **implementing** an **interface**

Polymorphism
Example

```
public abstract class Animal {
    public abstract String speak();
    public abstract void eat(double calories);
}
```

```
public class Dog extends Animal {
    public Dog () { }
    public String speak() {
        return "woof";
    }

    public void eat(double calories) {
        …
    }

    public Person getOwner()  { … }
}
```

```
public class Cow extends Animal {
    public Cow() { }
    public String speak() {
        return "moo";
    }

    public void eat(double calories) {
        …
    }

    public void getMilk()  { … }
}
```

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();
```

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;
```

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;
```

The Cow object takes the form of an animal (**Polymorphism**)

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();
```

The Cow object takes the form of an animal (**Polymorphism**)

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();

myCow.speak();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();

myCow.speak();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

A cow is also an animal, so it can speak

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();

myCow.speak();

myCow.getMilk();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

A cow is also an animal, so it can speak

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();

myCow.speak();

myCow.getMilk();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

A cow is also an animal, so it can speak

Cows give milk

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();

myCow.speak();

myCow.getMilk();

myAnimal.getMilk();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

A cow is also an animal, so it can speak

Cows give milk

Polymorphism
Example (2)

```
Cow myCow = new Cow();

Dog myDog = new Dog();

Animal myAnimal = myCow;

myAnimal.speak();

myCow.speak();

myCow.getMilk();

myAnimal.getMilk();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak

A cow is also an animal, so it can speak

Cows give milk

But animals can't! Even though this object is **actually a cow (Compilation Error)**

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak( ????  animal) {
        ????
}
```

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
        ????
}
```

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}
```

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();
```

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
```

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
```

moo

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
```

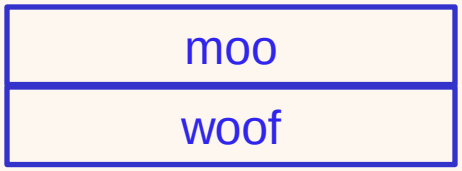
moo

Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
```

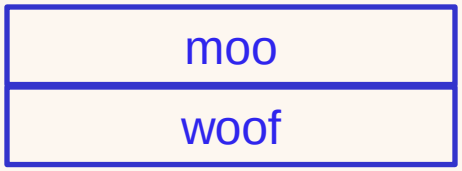


Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
makeAnimalSpeak(myAnimal);
```

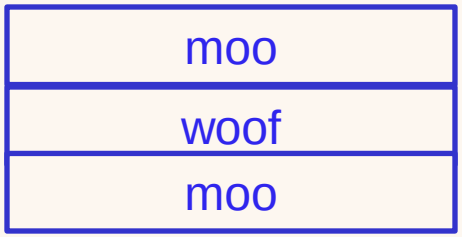


Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
makeAnimalSpeak(myAnimal);
```



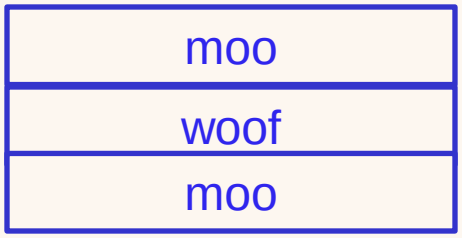
Polymorphism is Useful

```
/**
 * A function that get an animal argument of any type and make a sound.
 */
public void makeAnimalSpeak(Animal animal) {
    animal.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
makeAnimalSpeak(myAnimal);
```

It's the concrete object that counts!



Polymorphism is Useful (2)

```
/**
 * Get an array of animals and makes each of them make a sound.
 */
public void makeAnimalsSpeak( ???? l[ ] animals) {

                    ????

}
```

Polymorphism is Useful (2)

```
/**
 * Get an array of animals and makes each of them make a sound.
 */
public void makeAnimalsSpeak(Animal[ ] animals) {

                    ????

}
```

Polymorphism is Useful (2)

```
/**
 * Get an array of animals and makes each of them make a sound.
 */
public void makeAnimalsSpeak(Animal[ ] animals) {
    for (Animal animal: animals) {
            animal.speak();
    }
}
```

Polymorphism is Useful (2)

```
/**
 * Get an array of animals and makes each of them make a sound.
 */
public void makeAnimalsSpeak(Animal[ ] animals) {
    for (Animal animal: animals) {
            animal.speak();
    }
}

Animal[] animals = new Animal[3];
animals[0] = myDog;
animals[1] = myCow;
animals[2] = myAnimal;
makeAnimalsSpeak(animals);
```

Polymorphism is Useful (2)

```
/**
 * Get an array of animals and makes each of them make a sound.
 */
public void makeAnimalsSpeak(Animal[ ] animals) {
    for (Animal animal: animals) {
        animal.speak();
    }
}

Animal[] animals = new Animal[3];
animals[0] = myDog;
animals[1] = myCow;
animals[2] = myAnimal;
makeAnimalsSpeak(animals);
```

Polymorphism and Extensibility

- Polymorphism is so important because it allows us to build a program that is easy to extend

- Say now we build a new Animal class

```
class Goat extends Animal { … }
```

- Goat is an Animal, and thus it follows the Animal class API

- Consequently, makeAnimalsSpeak() will continue to work perfectly for objects of this class as well
  - Our program is **easy to extend**

Polymorphism and Flexibility

- Polymorphism allows us to modify the reference type during runtime

```
Animal myAnimal = new Cow();
myAnimal.speak();
```

Polymorphism and Flexibility

- Polymorphism allows us to modify the reference type during runtime

```
Animal myAnimal = new Cow();
myAnimal.speak();
```

moo

Polymorphism and Flexibility

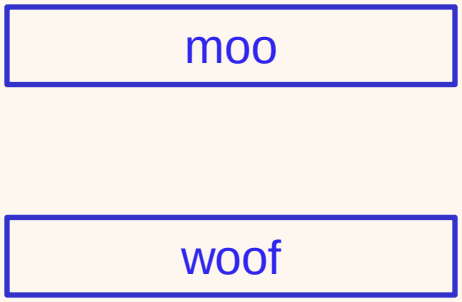
- Polymorphism allows us to modify the reference type during runtime

```
Animal myAnimal = new Cow();
myAnimal.speak();
…
myAnimal = new Dog();
myAnimal.speak();
```

Polymorphism and Flexibility

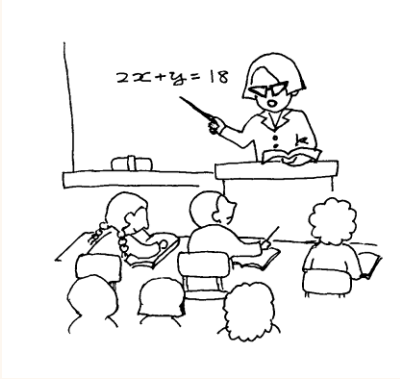
- Polymorphism allows us to modify the reference type during runtime

```
Animal myAnimal = new Cow();
myAnimal.speak();
…
myAnimal = new Dog();
myAnimal.speak();
```

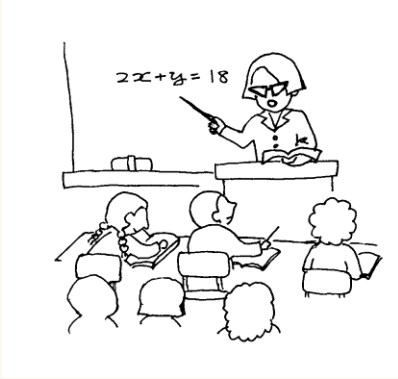


Interfaces Revised

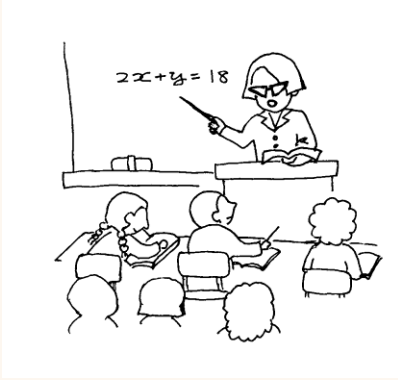
Interfaces Revised



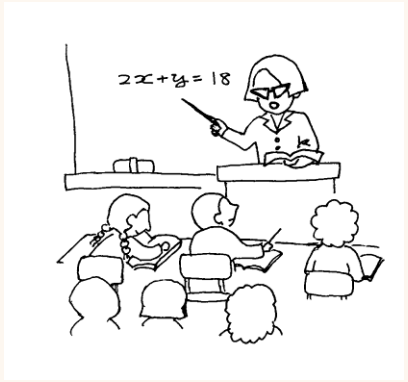
Interfaces Revised



Interfaces Revised



Interfaces Revised



And now to Java…

public class Double extends Number **implements** Comparable,Clonable

And now to Java…

```
public class Double extends Number implements Comparable,Clonable

/**
   * Solve a linear equation
   * @returns y = ax + b
   */
public Number solveLinearEquation(Number a, Number b, Number x) {…}
```

And now to Java…

```
public class Double extends Number implements Comparable,Clonable

/**
   * Solve a linear equation
   * @returns y = ax + b
   */
public Number solveLinearEquation(Number a, Number b, Number x) {…}

/** Apply bubble sort algorithm on arrayToSort. */
public void bubbleSort(Comparable[ ] arrayToSort) { … }
```

And now to Java…

```
public class Double extends Number implements Comparable,Clonable

/**
   * Solve a linear equation
   * @returns y = ax + b
   */
public Number solveLinearEquation(Number a, Number b, Number x) {…}

/** Apply bubble sort algorithm on arrayToSort. */
public void bubbleSort(Comparable[ ] arrayToSort) { … }

/** Create an array of n version of toClone. */
public Clonable[ ] cloneNTimes(Clonable toClone, int n) { … }
```

How does it Work?

```
public class Double extends Number implements Comparable,Clonable

public class String implements Comparable

Comparable[ ] array1 = new Double[] { … };

Comparable[ ] array2 = new String[] { … };


// Sort array1 and array2
bubbleSort(array1);
bubbleSort(array2);
```

How does it Work?

public class Double extends Number **implements** Comparable,Clonable

public class String implements Comparable

```
Comparable[ ] array1 = new Double[] { … };

Comparable[ ] array2 = new String[] { … };
```

```
// Sort array1 and array2
bubbleSort(array1);
bubbleSort(array2);
```

bubbleSort() doesn't care about the concrete type!
All that matters is that it **implements** Comparable

Which Method Runs?

- When we call myObj.foo(), it's the concrete type of foo that is called
  - myAnimal.speak() // Moo for cows, woof for dogs
  - This is also true if the parent class has a concrete implementation

Which Method Runs?

- When we call myObj.foo(), it's the concrete type of foo that is called
  - myAnimal.speak() // Moo for cows, woof for dogs
  - This is also true if the parent class has a concrete implementation

- On the other hand, we are only allowed to call methods from the **reference type**
  - myAnimal = **new** Cow(…);
  - myAnimal.getMilk() **// Compilation error. Animals // cannot give milk**

Polymorphism and Minimal API

- Recall: *when delivering a program, we want to share as few details as possible*
  - **Minimal API**

Polymorphism and Minimal API

- Recall: *when delivering a program, we want to share as few details as possible*
  - **Minimal API**

- We saw this principle when discussing the **private** modifier
  - **Information Hiding**

Polymorphism and Minimal API

- Recall: *when delivering a program, we want to share as few details as possible*
  - **Minimal API**
- We saw this principle when discussing the **private** modifier
  - **Information Hiding**
- Polymorphism allows us to take this principle a step further
  - **Program to interface, not to implementation**

Program to interface, not to implementation

- When defining an API, we should attempt at using types higher at the class hierarchy
- If our code only uses the API of the higher type, there is no reason to use more concrete classes
  - makeAnimalSpeak(**Animal** animal) and **not** makeCowSpeak(**Cow** cow)

Program to interface, not to implementation (2)

- This principle has several benefits
  - **Generality**: the same code works for more objects
    - All Animals and not just Cows
  - **Extensibility**: adding a new type (**class** Goat **extends** Animal) is automatically supported by this code
  - **Easier modification**: clients remain unaware of the specific type of objects they use
    - This Allows replacing implementation without affecting clients
    - This greatly reduces implementation dependencies between subsystems
  - **Code is easier to learn**: there are fewer higher level types than lower level ones

So far…

- Polymorphism
  - Extendibility
  - Flexibility
  - Interfaces
  - Information hiding (Program to **interface**, not to **implementation**)

Casting

- Casting is the operation in the heart of polymorphism: referring to an object of one type with a different reference type
  - Animal a = **new** Cow();
  - This type of casting is called **up-casting**

Casting

- Casting is the operation in the heart of polymorphism: referring to an object of one type with a different reference type
  - Animal a = **new** Cow();
  - This type of casting is called **up-casting**

- Up-casting is allowed iff the reference (i.e., the **left-hand** side) is a super class (or interface) of the concrete object (**right-hand** side)

Down Casting

- A more complicated type of casting is **down-casting**
- Down-casting is the operation of assigning a reference with a class which is **not** a sub-class (or an implementing class) of the reference type

Down Casting

- A more complicated type of casting is **down-casting**
- Down-casting is the operation of assigning a reference with a class which is **not** a sub-class (or an implementing class) of the reference type
- This operation requires a special syntax:
  - Animal animal = …;
  - Cow c = **(Cow)** animal;

Down Casting

- A more complicated type of casting is **down-casting**
- Down-casting is the operation of assigning a reference with a class which is **not** a sub-class (or an implementing class) of the reference type
- This operation requires a special syntax:
  - Animal animal = …;
  - Cow c = **(Cow)** animal;

Down Casting (2)

- Down-casting can sometimes succeed
  - if the left-hand side's **real** type is actually a sub-class (or implementing class) of the reference type
  - Animal animal = **new** Cow()
  - Cow c = **(Cow)** animal;

Down Casting (2)

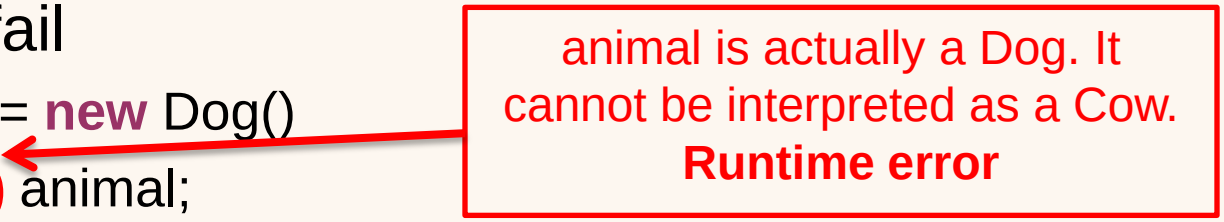
- Down-casting can sometimes succeed
  - if the left-hand side's **real** type is actually a sub-class (or implementing class) of the reference type
  - Animal animal = **new** Cow()
  - Cow c = **(Cow)** animal;

- But it can also fail
  - Animal animal = **new** Dog()
  - Cow c = **(Cow)** animal;
  - Cow c2 = **(Cow)** **new** Integer(5);

Down Casting (2)

- Down-casting can sometimes succeed
  - if the left-hand side's **real** type is actually a sub-class (or implementing class) of the reference type
  - Animal animal = **new** Cow()
  - Cow c = **(Cow)** animal;

- But it can also fail
  - Animal animal = **new** Dog()
  - Cow c = **(Cow)** animal;
  - Cow c2 = **(Cow) new** Integer(5);



Down Casting (2)

- Down-casting can sometimes succeed
  - if the left-hand side's **real** type is actually a sub-class (or implementing class) of the reference type
  - Animal animal = **new** Cow()
  - Cow c = **(Cow)** animal;

- But it can also fail
  - Animal animal = **new** Dog()
  - Cow c = **(Cow)** animal;
  - Cow c2 = **(Cow)** **new** Integer(5);

animal is actually a Dog. It cannot be interpreted as a Cow. **Runtime error**

Cow is not a sub-class of Integer. This operation can never succeed. **Compilation error**

We don’t Like Down Casting

- An object of type C can potentially be cast to any class that **extends** / **implements** C
  - Such code will always compile
- However, casting can fail at **run-time**
- Run-time errors are **very expensive**
  - Time, reputation, money, …
- There is almost always a better alternative than using down casting

We don’t Like Down Casting (2)

- Remember flexibility?
  - makeAnimalsSpeak(…) will continue to work even after writing a new Goat class

- Using down casting, this is no longer correct
  - We are hard-coding specific class names, thus making our code **fixed** and not **flexible**

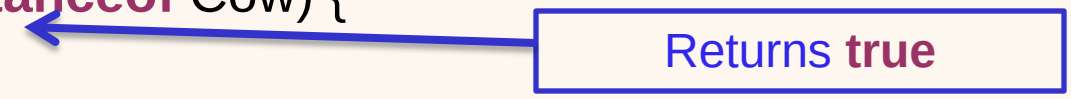
instanceof

- **instanceof** is a java operator that allows us to check whether an object is an instance of a given class
  - Animal animal = **new** Cow()
  - **if** (animal **instanceof** Cow) {

- Supposedly, **instanceof** could be used as a safety measure against the runtime errors previously presented

instanceof

- **instanceof** is a java operator that allows us to check whether an object is an instance of a given class
  - Animal animal = **new** Cow()
  - **if** (animal **instanceof** Cow) {



- Supposedly, **instanceof** could be used as a safety measure against the runtime errors previously presented

We also don’t Like instanceof

- Using **instanceof** is still bug prone
- Also, **instanceof** code is inflexible
- Using **instanceof** is generally considered **bad practice**
  - Although there are exceptions

Bad instanceof Example

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
      if (animal instanceof Fish) {
            ((Fish)animal).swim();
      } else if (animal instanceof Horse) {
            ((Horse)animal).ride();
      }
      ...
    }
}
```

Bad instanceof Example

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
      if (animal instanceof Fish) {
            ((Fish)animal).swim();
      } else if (animal instanceof Horse) {
            ((Horse)animal).ride();
      }
      ...
    }
}
```

- What happens if we want to support a new animal (snake)?

Bad instanceof Example

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
      if (animal instanceof Fish) {
            ((Fish)animal).swim();
      } else if (animal instanceof Horse) {
            ((Horse)animal).ride();
      }
      ...
    }
}
```

- What happens if we want to support a new animal (snake)?
  - Code has to change
  - Bugs may arise

instanceof Alternative

- Use a common API
  - Animal.move()

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
            animal.move();
    }
}
```

instanceof Alternative

- Use a common API
  - Animal.move()

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
            animal.move();
    }
}
```

- *Simpler, shorter, safer, easier to extend*

Casting Examples

Animal a; Cow c; Dog d;

a	c	d

Casting Examples

Animal a; Cow c; Dog d;

d = **new** Dog();

a	c	d

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
```

a	c	d
		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();    // OK
a = new Cow(5);
```

a	c	d
		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
a = new Cow(5);     // OK (up-casting)
```

a	c	d
Cow		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
a = new Cow(5);     // OK (up-casting)
a.speak();
```

a	c	d
Cow		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
a = new Cow(5);     // OK (up-casting)
a.speak();          // Returns “moo”
```

a	c	d
Cow		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
a = new Cow(5);     // OK (up-casting)
a.speak();          // Returns “moo”
a = d;
```

a	c	d
Cow		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
a = new Cow(5);     // OK (up-casting)
a.speak();          // Returns “moo”
a = d;              // OK (up-casting)
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();      // OK
a = new Cow(5);     // OK (up-casting)
a.speak();          // Returns “moo”
a = d;              // OK (up-casting)
a. speak();         // Returns “woff”
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();        // OK
a = new Cow(5);       // OK (up-casting)
a.speak();            // Returns “moo”
a = d;                // OK (up-casting)
a. speak();           // Returns “woff”
d = (Dog) a;          // OK  (down-casting)
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (down-casting without casting operation)
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (down-casting without casting operation)
c = (Cow) a;
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (down-casting without casting operation)
c = (Cow) a;            // Run-time error (incompatible down casting)
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (down-casting without casting operation)
c = (Cow) a;            // Run-time error (incompatible down casting)

if (a instanceof Cow) {
    c = (Cow) a;
}
```

a	c	d
Dog		Dog

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (up-casting)
a.speak();              // Returns “moo”
a = d;                  // OK (up-casting)
a. speak();             // Returns “woff”
d = (Dog) a;            // OK  (down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (down-casting without casting operation)
c = (Cow) a;            // Run-time error (incompatible down casting)

if (a instanceof Cow) {
    c = (Cow) a;        // OK (given the instanceof check result),
}                       // though not recommended
```

a	c	d
Dog	Cow?	Dog

Reuse Mechanisms

- What is the right way to build reusable software?

Reuse Mechanisms

- What is the right way to build reusable software?

- **Inheritance** provides a built-in mechanism for sharing code
  - Extending a class gives us access to all its **public** and **protected** members and methods
  - This is considered by many one of the major reasons for using a class hierarchy

Reuse Mechanisms

- What is the right way to build reusable software?

- **Inheritance** provides a built-in mechanism for sharing code
  - Extending a class gives us access to all its **public** and **protected** members and methods
  - This is considered by many one of the major reasons for using a class hierarchy

- However, there is an alternative mechanism to code reuse
  - **Object composition**

Reuse Mechanisms

Inheritance

- Define an implementation of one class in terms of another's

- Called “**white-box**” reuse since the internals of the parent are visible to its subclasses

Reuse Mechanisms

Inheritance example

```
public class B {
    protected void foo() { … }
}

public class A extends B {
    …
}
```

- Now, A got the *foo*() method for free and can use it

Reuse Mechanisms

Composition

- An alternative to class inheritance
- New functionality is obtained by assembling (or composing) objects to get more complex functionality
- Requires that objects being composed have well defined interfaces
- Called **“black-box”** reuse because no internal details of objects are visible
  - Objects appear only as “black boxes”

Reuse Mechanisms

Composition example

```
public class B {
      public void foo() { … }
}

public class A {
     private B b;

     public A(B b) {
           this.b = b;
     }

     public anotherFoo(…) {
           …
           this.b.foo();          // A uses the foo() code by calling b.foo()
           …
     }
}
```

Reuse Mechanisms

Composition example

```
public class B {
      public void foo() { … }
}

public class A {
     private B b;

     public A(B b) {
           this.b = b;
     }

     public anotherFoo(…) {
           …
           this.b.foo();          // A uses the foo() code by calling b.foo()
           …
     }
}
```

Inheritance

- **Pros of inheritance:**
  - Straightforward to use (supported by the programming language): enables **polymorphism**
  - Defined statically at compile time
  - Easier to modify the implementation being reused by overriding

- **Cons of inheritance:**
  - An implementation cannot be changed at runtime
  - Breaks encapsulation – a subclass is exposed to details of the parent's implementation (**protected** members/methods)
  - Subclass implementation is bound to parent implementation
  - Any change in the parent forces the subclass to change
  - A class can only extend a single parent-class

Object Composition

- **Pros of object composition:**
  - Defined dynamically at runtime
  - Encapsulation is not broken and therefore any object can be replaced at runtime by another as long as it has the same type
  - Substantially fewer implementation dependencies
  - Helps keep each class encapsulated and focused on one task
  - A class may compose as many objects as it wants

- **Cons of object composition:**
  - A composition based design has more objects (if fewer classes)
  - No support for **polymorphism**
  - The system's behavior will depend on their interrelationships instead of being defined in one class

Reuse Mechanisms

Inheritance vs. Composition

- Use *inheritance* when:
  - A is **inherently** a B
    - A dog **is** an animal
- Use *object composition* when:
  - You mainly want to **reuse code**
  - When it makes more sense to extend A by another class C
- If you only need **polymorphism** (but an A **is not** a B), consider using *interfaces*
- *Inheritance* and object *composition* **work together**

So far…

- Polymorphism
- Casting
- Reuse Mechanism
  - Inheritance vs. composition