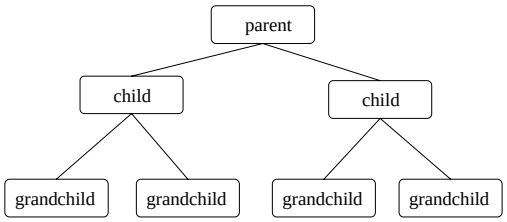
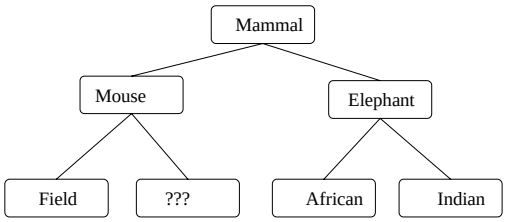
Chap 7 - Inheritance



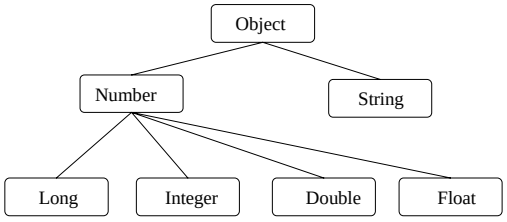
1

An example from Zoology



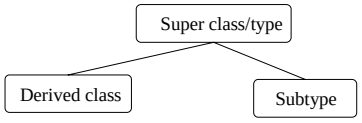
2

An example from java.lang



3

Other terms from inheritance



4

```
//Person.java - characteristics common to all people
class Person {
  Person(String name)        { this.name = name; }
  void setAge(int age)       { this.age = age; }
  void setGender(char gender){ this.gender = gender; }
  void setName(String name)  { this.name = name; }
  int getAge()               { return age; }
  char getGender()           { return gender; }
  String getName()           { return name; }
  public String toString() {
    return ("Name: " + name +
      ", Age: " + age + ", Gender: " + gender);
  }
  private String name;
  private int age;
  private char gender; //male == 'M' , female == 'F'
}
```

5

```
//Student.java - an example subclass
class Student extends Person  {
  Student(String name) {  super(name); }
  void setCollege(String college)
      { this.college = college; }
  void setGpa(double gpa)   { this.gpa = gpa; }
  void setYear(byte year)   { this.year = year; }
  String getCollege()       { return college; }
  double getGpa()           { return gpa; }
  byte getYear()            { return year; }
  public String toString() {
    return(super.toString() + "\n  " +
      "College: " + college +
      ", GPA: " + gpa +
      ", Year: " + year);
  }
```

6

```
  static final byte FROSH = 1;
  static final byte SOPH = 2;
  static final byte JUNIOR = 3;
  static final byte SENIOR = 4;
  private String college = "Unknown";
  private byte year; // FROSH, SOPH, ...
  private double gpa; //0.0 to 4.0
}
```

7

```
//StudentTest.java
class StudentTest {
  public static void main(String[] args) {
    Student student = new Student("Jane Programmer");

    student.setAge(21);
    student.setGender('F');
    student.setCollege("UCSC");
    student.setYear(Student.FROSH);
    student.setGpa(3.75f);
    System.out.println(student.toString());
  }
}
```

8

Subtype Principle

- Methods defined in one class may be redefined in a subclass. This is called method overriding.
- *A subclass object can always be used where an object of its superclass is expected.*
- Treating a subclass object as a superclass object can only remove capabilities, not add them. With inheritance, new methods are added in the subclass, never taken away.

9

Which toString() gets called?

```
class StudentTest {
  public static void main(String[] args) {
    Student student = new Student("Jane Programmer");
    student.setAge(21);
    student.setGender('F');
    student.setCollege("UCSC");
    student.setYear(Student.FROSH);
    student.setGpa(3.75f);
    System.out.println(student.toString());
    Person anyPerson = studetn;
    System.out.println(anyPerson.toString());
  }
}
```

10

Can't use Student fields from Person reference

```
class StudentTest {
  public static void main(String[] args) {
    Student student = new Student("Jane Programmer");
    student.setAge(21);
    ...
    System.out.println(student.toString());
    Person anyPerson;
    if (Console.in.readInt() == 1)
        anyPerson = student;
    else
        anyPerson = new Person("John Doe");
    anyPerson.setYear(Student.FROSH); // illegal
    System.out.println(anyPerson.toString());
  }
}
```

11

Overriding vs Overloading

- Overloading is when you define two methods with the same name, in the same class, distinguished by their signatures.
- Overriding is when you redefine a method that has already been defined in a parent class (using the exact same signature).
- Overloading is resolved at compile time.
- Overriding is resolved at runtime (based on the type of the implicit first parameter).

12

Dynamic Method Dispatch

- Determing at runtime, which overriden method to call, is called *dynamic method dispatch.*
- This is what allows println() to work with any object.
- toString() is defined in Object (the parent of all classes).
- If you override toString(), then inside of println(), a call to printThis.toString(), will get to YOUR toString().

13

```
public class PrintWriter {
    ...
    public void println(Object obj) {
        String s = obj.toString();
        // somehow get the string s printed
    }
}
```

14

Dynamic Method Dispatch

```
//SuperClass.java - a sample super class
class SuperClass {
  public void print() {
    System.out.println( " inside SuperClass");
  }
}
//SubClass.java - a subclass of SuperClass
class SubClass extends SuperClass {
  public void print() {
    System.out.println( " inside SubClass");
  }
}
```

15

```
//TestInherit.java - overridden method selection.
class TestInherit {
  public static void main(String[] args) {
    SuperClass s =  new SuperClass();
    s.print();
    s = new SubClass();
    s.print();
  }
}
```

16

Access Modifiers, Method Overriding and the Subtype Principle

- When you override a method you cannot restrict access more than was done with the inherited method.
- Doing so would break the subtype principle.
- For example, if class Student overrode toString() to be private, then `System.out.println(anyPerson.toString());` would fail if anyPerson was referring to a Student.

17

Generic Methods

- A *generic method* is a method that can operate on at least two different types of data. This is a form of *polymorphism*.
- println(Object obj) is a generic method. It works for any Java reference type.
- In addition to toString(), the class Object defines a method
  `public boolean equals(Object obj);`

18

Method equals() in Object

- By overriding equals() we can write a generic methods that need to check if two objects are equal.


```
//Count the number of times obj is found in array.
static int howManyCopiesOf(Object obj,
                           Object[] array) {
  int  count = 0;

  for (int i = 0; i < array.length; i++)
     if (obj.equals(array[i]))
         count++;
  return count;
}
```



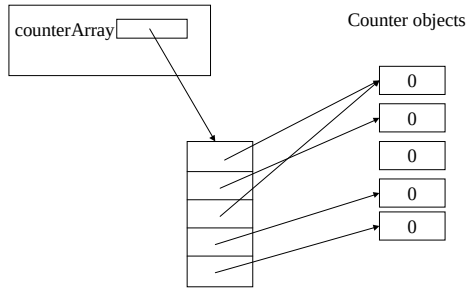
```
class EqualsTest {
  public static void main(String[] args) {
    // Create and fill an array of Strings
    String[] stringArray = new String[10];
    for (int i = 0; i < stringArray.length; i++)
      stringArray[i] = "String " + i;
    // Create and fill an array of Counters
    Counter[] counterArray = new Counter[5];
    for (int i = 0; i < counterArray.length; i++)
      counterArray[i] = new Counter();
    // Make two entries refer to the same Counter
    counterArray[2] = counterArray[0];

    System.out.println(
      howManyCopiesOf(counterArray[0], counterArray));
    System.out.println(
      howManyCopiesOf("String 1", stringArray));
  }
```


When equals are not equal?

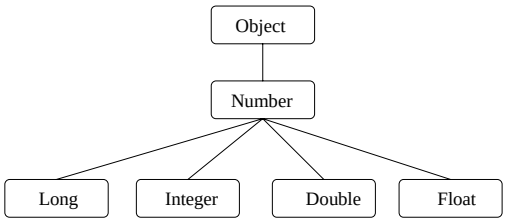
- Using the original Counter from chapter 6, the output of the previous program is 2 and 1. That's 2 Counters "equal" to counterArray[0], and 1 String "equal" to "String 1".
- The class String, overrides equals() to compare the characters in two Strings.
- Counter uses the default equals() which just checks if the references point to the same object.


```
counterArray[2] = counterArray[0];
```




```
counterArray[2].equals(counterArray[0]) is true
```


Primitive Wrapper Classes



A Generic Numeric Method

```
static Number elementMin(Number[] array) {
  Number min = array[0];

  for (int i = 1; i < array.length; i++)
    if (array[i].doubleValue() < min.doubleValue())
      min = array[i];
  return min;
}
```

This works so long as the loss of precision in coverting to double doesn't affect the selected minimum.

25

Abstract Classes

- An abstract class is use to derive other (concrete) classes.
- An abstract class usually provides some complete method definitions, but leaves some undefined, requiring all subclasses to provide definitions for the undefined methods.

26

```
abstract public class AbstractCounter {

  abstract public void click();
  public int get() { return value; }
  public void set(int x) { value = x; }

  public String toString()
            { return String.valueOf(value); }

  protected int value;
}

public class Counter extends AbstractCounter {
  public void click() { value = (value + 1) % 100; }
}
```

27

```
public class CountByTwo extends AbstractCounter {
  public void click() { value = (value + 2) % 100; }
}

void sampleMethod(AbstractCounter counter) {
  ...
  counter.click();
  ...
}
```

- You can declare variables/parameters of abstract types.
- You cannot ever actually create an object of an abstract class.
- E.g. new AbstractCounter() is illegal.

28

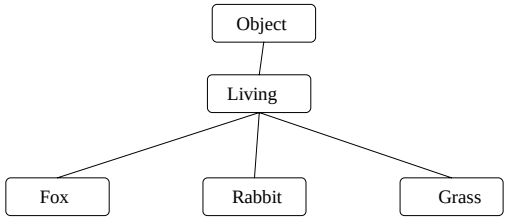
```
class Timer extends AbstractCounter {
  public Timer(int v) { set(v); }
  public void click() {
    value++;
    seconds = value % 60;
    minutes = value / 60;
  }
  public void set(int v) {
    value = v;
    seconds = v % 60;
    minutes = v / 60;
  }
  public String toString() {
    return minutes + " minutes, " +
           seconds + " seconds";
  }
  private int seconds, minutes;
}
```

You can, of course, override any of the methods inherited from an abstract class.

29

Predator-Prey: An abstract class

A simple simulation of an artificial ecology



30

Pseudocode for Predator-Prey

- create two worlds, current and next
- initialize one with some life forms
- print the intial world
- for each step of the simulation
  - update next based on current
  - print the next world
  - swith the roles of current and next

31

```
class PredatorPrey {
  public static void main(String[] args) {
    World odd = new World(10), even = new World(10);
    int i, cycles = 10;
    even.eden();              //generate initial World
    System.out.println(even); //print initial state
    for (i = 0; i < cycles; i++) {
      System.out.println("Cycle = " + i + "\n\n");
      if (i % 2 == 1) {
        even.update(odd);
        System.out.println(even);
      }
      else {
        odd.update(even);
        System.out.println(odd);
      }
    }
  }
}
```

32

Counting Neighbors

- Life and death of life forms depends upon the number of various life forms in adjacent cells. Doing this counting is the same for all life forms so we will implement it in an abstract class, Living.
- Pseudocode
  - set the count for all life form types to 0
  - for each of the current cells 8 immediate neighbors
    - if the neighbor is type LifeType
    - then increment the count for LifeType

33

```
//Living.java - the superclass for all life forms
abstract class Living {

  abstract Count getCount();
  abstract Living next(World world);
  abstract char toChar(); // character for this form
  void computeNeighbors(World world) {
    world.clearNeighborCounts();
    world.cells[row][column].getCount().set(-1);
    for (int i = -1; i <= 1; i++)
      for (int j = -1; j <= 1; j++)
        world.cells[row+i][column+j].getCount().inc();
  }
  int row, column; //location
}
```

34

```
class Fox extends Living {
  Fox(int r, int c, int a )
    { row = r; column = c; age = a; }
  Living next(World world) {
    computeNeighbors(world);
    if (Fox.neighborCount.get() > 5 ) //too many Foxes
      return new Empty(row, column);
    else if (age > LIFE_EXPECTANCY)   //Fox is too old
      return new Empty(row, column);
    else if (Rabbit.neighborCount.get() == 0)
      return new Empty(row, column);  // starved
    else
      return new Fox(row, column, age + 1);
  }
```

35

```
  public String toString(){ return "Fox age " + age; }
  char toChar() { return 'F'; }
  Count getCount() {  return neighborCount; }
  static Count neighborCount = new Count();

  private int age;
  private final int LIFE_EXPECTANCY = 5;
}
```

36

Why must getCount() and neighborCount be repeated in each subclass of Living? Why not just move these definitions to Living?

Answer: There is no way to write a method in Living, that accesses a static field in the subclasses of Living. We need a neighborCount for each of Fox, Rabbit, Grass, and Empty.

```
class Rabbit extends Living {
  Rabbit(int r, int c, int a )
    { row = r; column = c; age = a;}
  ...
  Count getCount() { return neighborCount; }

  static Count neighborCount = new Count();
  private int age;
  private final int LIFE_EXPECTANCY = 3;
}
```


Interfaces

- An interface in Java, is essentially an pure abstract class - all methods are abstract.
- An interface can contain only abstract methods, and constants. There are no instance variables.

Suppose that different life forms have different notions of a neighborhood. Then in our simulation we can't provide computeNeighbors() that works for all. We might just go with an interface instead of an abstract class.

```
interface Living {
  Living next(World world);
  char toChar();
  void computeNeighbors(World world);
  Count getCount();
}
```



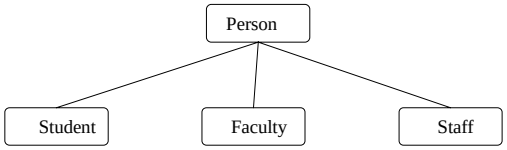
```
class Fox implements Living {
  // omitted methods/instance variables that are the
  // same as the earlier Fox class.

  public void computeNeighbors(World world) {
    // each class must now define this method
  }

  // an interface doesn't contain data fields
  // so row and column must be declared here
  // we can now make these members private
  private int row, column;
}
```

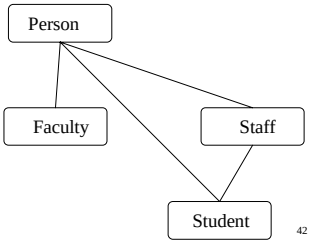

Multiple Inheritance

- What happens if a student is also a staff member?



Multiple Inheritance

- What happens if a student is also a staff member?



implements vs extends

- Java does not allow you to extend multiple classes. This avoids some problems related to multiple inheritance (e.g. what happens if you inherit a method from two different parent classes?).
- Java does allow you to implement multiple interfaces.
- Interfaces are about subtyping and polymorphism, whereas, inheriting methods is about code reuse.

43

```
interface Person {
  void setAge(int age);
  void setGender(char gender);
  void setName(String name);
  int getAge();
  char getGender();
  String getName();
}
```

44

```
interface Student extends Person {
  void setCollege(String college);
  void setGpa(double gpa);
  void setYear(byte year);
  String getCollege();
  double getGpa();
  byte getYear();
  static final byte FROSH = 1;
  static final byte SOPH = 2;
  static final byte JUNIOR = 3;
  static final byte SENIOR = 4;
}
```

45

```
import java.util.Date;

interface Staff extends Person {
  void setSalary(double salary);
  void setStartDate(Date start);
  void setEndDate(Date end);
  void setSSN(String ssn);
  double getSalary();
  Date getStartDate();
  Date getEndDate();
  String getSSN();
}
```

46

```
import java.util.Date;

class StudentEmployee implements Student, Staff {
  // methods required by Person
  public void setAge(int age){... }
  ...
  public String getName(){...}
  // methods required by Student
  public void setCollege(String college){... }
  ...
  public byte getYear(){...}
  // methods required by Staff
  public void setSalary(double salary){...}
  ...
  public String getSSN(){...}
}
```

47

Can't instantiate an interface

- Just as with abstract classes, you can declare variables and parameters to be of an interface type, but you cannot create objects of an interface type.
- Using the declartions on the previous slides, **new Staff()** would be illegal.
- You could declare a variable or parameter to be of type Staff. This variable would always either be null or refer to an instance of some class that implements the interface Staff.

48

instanceof

```
if (x intanceof Shape)
    // x is a Shape, act accordingly
else
    // x is not a Shape, act accordingly
...
Person person = new Student();
...
if (person instanceof Student) {
    Student student = (Student)person;
    // operate on student
}
```

49

Illegal Cast

- Some illegal casts can be determined at compile time, others can only be detected at runtime.
- Trying to cast a Person into a String is a compile time error. The compiler knows that String is not a subclass of Person and hence the cast is always illegal.
- Trying to cast a Person into a Student, when the Person is not a Student, is a runtime error - even if YOU can see that this person couldn't possibly be a Student.

50

ClassCastException

- The following can only be detected at runtime, even though you can see it is clearly illegal.

```
Person person = new Person();
Student student = (Student)person;
```

51

```
// GenericArray.java -
//     demonstrate generic array container
class GenericArray {
  public static void main(String[] args) {
    Object[] array = new Object[4];

    array[0] = "String 1";
    array[1] = new Integer(1);
    array[2] = "String 2";
    array[3] = new Integer(2);
    for (int i = 0; i < array.length; i++) {
      // see next slide for the body of the loop
    } // end of for loop
  }
}
```

52

```
if (array[i] instanceof String) {
  String temp = (String)array[i];
  System.out.println("Processing string "
      + temp);
  // do something appropriate for strings
}
else if (array[i] instanceof Integer) {
  Integer temp = (Integer)array[i];
  System.out.println("Processing Integer "
      + temp);
  // do something appropriate for an Integer
}
else {
  System.out.println("Unexpected type "
      + array[i]);
  // do something to handle unexpected cases
}
```

53