An Example Base Class

```
class Person
{
    public Person( String n, int ag, String ad, String p )
         { name = n; age = ag; address = ad; phone = p; }

    //accesspr (getter) methods
    public String getName( ) { return name; }
    public int getAge( ) { return age; }
    public String getAddress( ) { return address; }
    public String getPhoneNumber( ) { return phone; }

    //mutator (setter) methods
    public void setAddress( String newAddress )
        { address = newAddress; }
    public void setPhoneNumber( String newPhone )
        { phone = newPhone; }

    public String toString() {return "Name: "+getName()+
             ", Age: "+getAge()+", Address: "+getAddress()+
             ", Phone: "+getPhone(); }

    private String name, address, phone;
    private int age;
}
```

Public methods

Private instance variables

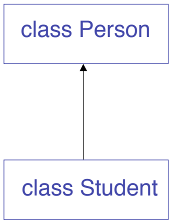
Create a "derived" student class

- A student is a type of person
- Add a couple of fields and methods just for students:
  - **`gpa`** field
  - **`getGPA`** accessor
- Using inheritance, can say a student IS-A person, then specify modifications
- Derived class must specify its own constructors

Accessors (Getters) are methods that access one (private) field in a class. They typically have names starting with "get". Mutators (Setters) are methods that modify one (private) field in a class. They typically have names starting with"set".

"family trees" of Classes

Conceptually, we can look at the class inheritance as a tree (like a family tree) called a class diagram



class Person is the superclass (base, parent, ancestor) of Student

class Student is a subclass (child, descendent) or derived class of Person

We say Student "inherits" certain fields or methods from Person

Modifications to Derived Classes

- Three types of modifications allowed:
  1. Add new fields (e.g., **`gpa`**)
  2. Add new methods (e.g., **`getGPA)`**
  3. Override existing methods (e.g., **`toString`**)

The Student Class : Preliminary Declaration

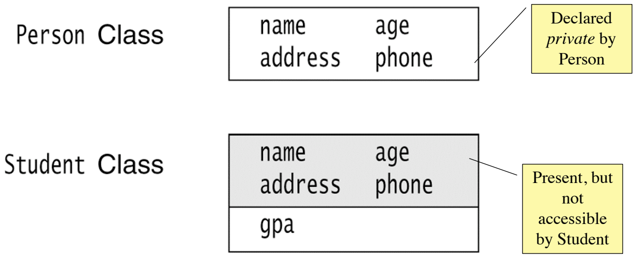
```
class Student extends Person
{
    private double gpa;

    public Student( String n, int ag, String ad, String p, double g )
    {
        // Need something more here in constructor …
        gpa = g;
    }

    public String toString( )
    {
        return getName() + " " + getGPA();
    }

    public double getGPA( )
    {
        return gpa;
    }
}
```

Add a data field to constructor

Override a base method

Add a method not in base class

Memory Layout with Inheritance



Light shading indicates fields that are private, and accessible only by methods of the class. Dark shading in the Student class indicates fields that are not accessible in the Student class (except by calling "getter" and "setter" methods in the Person class), but are nonetheless present.

Constructors for Derived Classes

- Each derived class should include constructors
  - If none present, a single zero-parameter constructor is generated
    - Calls the base class zero-parameter constructor
    - Applies default initialization for any additional fields defined in the derived class
- Good practice: call the superclass constructor first in derived class constructor

Recall that the default initialization is 0 for primitive types and null for reference types

The **super** Keyword

- **super** is the keyword used to explicitly call the base (superclass) constructors
- Default constructor for a derived class is really

```
public Derived()
{
      super();
}
```

- **super** method can be called with parameters that match the base class constructor

Example: Student, a derived class

```
class Student extends Person
{
    private double gpa;

    public Student( String n, int ag, String ad, String p, double g )
    {
        super( n, ag, ad, p );
        gpa = g;
    }

    public String toString( )
    {
        return super.toString( ) + " " + getGPA();
    }

    public double getGPA( )
    {
        return gpa;
    }

}
```

Calls superclass constructor with four parameters

Can call superclass methods using **super**

Partial overriding: use **super** to call a superclass method, when we want to do what the base class does plus a bit more, as in this example

Another derived class

```
class Employee extends Person
{
    private double salary;

    public Employee( String n, int ag, String ad, String p, double s )
    {
        super( n, ag, ad, p );
        salary = s;
    }

    public String toString( )
    {
        return super.toString( ) + " " + getSalary();
    }

    public double getSalary( )
    {
        return salary;
    }
}
```

Calls superclass constructor with four parameters

Can call superclass methods using **super**

Type Compatibility

- Because a **Student** IS-A **Person**, a **Student** object can be accessed by a **Person** reference

```
Student s = new Student ( “Joe”, 26, “1 Main St”,
                          “845-555-1212”, 4.0 );
Person p = s;
System.out.println( “age is “ + p.getAge() );
```

- **p** may reference any object that IS-A **Person**
- Any method in either the Person or Student class invoked through the **p** reference is guaranteed to work because methods defined for class **Person** cannot be removed by a derived type

Why is this a big deal?

- Because it applies not only to assignment, but also argument passing
  - I.e., a method whose formal parameter IS-A **Person** can receive any object that IS-A **Person**, such as **Student**

Consider this static method written in *any class*

```
public static boolean isOlder( Person p1, Person p2 )
{
      return p1.getAge() > p2.getAge();
}
```

Suppose some declarations are made... (arguments omitted)

```
Person    p = new Person (…)
Student   s = new Student (…)
Employee  e = new Employee (…)
```

Can use `isOlder` with all the following calls

```
isOlder(p,p), isOlder(s,s), isOlder(e,e),
isOlder(p,e), isOlder(p,s), isOlder(s,p),
isOlder(s,e), isOlder(e,p), isOlder(e,s),
```

For many, type compatibility of derived classes with the base class is the most important thing about inheritance because it leads to massive *indirect code reuse*

Dynamic Binding and Polymorphism

If the type of the reference (e.g., `Person`) and the class of the object being referenced (e.g., `Student`) disagree, and they have different implementations, whose implementation is used?

```
Student s = new Student ( “Joe”, 26, “1 Main St”,
                          “845-555-1212”, 4.0 );
Employee e = new Employee ( “Boss”, 42, “4 Main St”,
                          “854-555-1212”, 10000.0 );
Person p = null;
if( ((int)(Math.random() * 10)) % 2 == 0 )
      p = s;
else
      p = e;
System.out.println( “Person is “ + p );
```

Do not know until program runs whether to use `Student`’s `toString` or `Employee`’s `toString`

Polymorphism

- When we run the program, the *dynamic type* (i.e., the most *specific* type of the object being referenced) will determine which method is used

Static type : a type associated with an entity at compile-time (does not change at any time during program execution)

Dynamic type : a type associated with an entity at run-time (may change on subsequent executions of the same statement)

Exceptions

- Objects that store information that is transmitted outside the normal *return* sequence; not an intended part of the program
- Propagated back through the calling sequence until a routine *catches* the exception
- At this point, can use information in the object to provide *error handling*
- Used to signal exceptional occurrences such as errors
- System generates its own exceptions and you can write your own

You have already seen `java.lang.ArrayIndexOutOfBoundsException`

Catching Exceptions with try and catch

```
public class ExceptionTest
{
    public static void main( String [ ] args ){
        int numLines = 10;
        int currLine = 0;

        String[ ] array = getStrings( numLines );

        try {
          while( currLine <= numLines ){
            System.out.println( array[currLine++] );
          }
        }
        catch (ArrayIndexOutOfBoundsException msg)
        {
          System.out.println(currLine + “invalid index.”);
        }
    }
    public static String[ ] getStrings(int nLines )
        …
```

the code to check

the code to execute on exception

Throw Clause

- Programmer can generate an exception using keyword **`throw`**
- Can create new message to produce in cases where exceptions occur

Example

```
catch (ArrayIndexOutOfBoundsException aioobx)
{
   throw new TooManyPeopleException(
               "Not enough space for more people");
}
```

Defining Exceptions

- If you are throwing an exception that is not one of the built-in Java exceptions, you must declare it as a class in the same directory as the program that uses it and extend RunTimeException.

Example

```
public class TooManyPeopleException extends RunTimeException {
{
   public TooManyPeopleException(String msg) {
      super(msg);
   }
}
```

Throws Clause

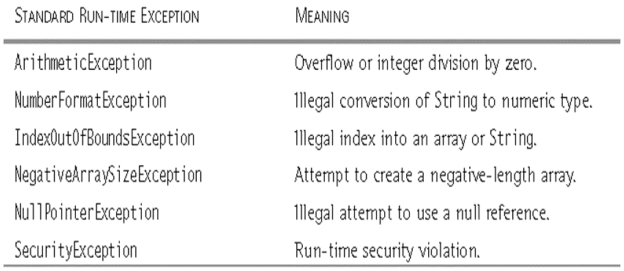
- Include **`throws`** clause when a method is declared that may generate an exception that is not derived from RunTimeException.

Common example when reading from and writing to files

```
public static void readFile() throws IOException
{
      ...
}
```

- We will see more on exceptions throughout the course.

Classes derived from RunTimeException



STANDARD RUN-TIME EXCEPTION	MEANING
ArithmeticException	Overflow or integer division by zero.
NumberFormatException	Illegal conversion of String to numeric type.
IndexOutOfBoundsException	Illegal index into an array or String.
NegativeArraySizeException	Attempt to create a negative-length array.
NullPointerException	Illegal attempt to use a null reference.
SecurityException	Run-time security violation.

Interfaces

- In order for objects to interact, they must "know" about the public methods each supports, (I.e., exports.)
- Java application programming interface (API) requires classes to specify the interface they present to other objects.
- The major structural element in Java that supports an API is the *interface*

==> Collection of method declarations and/or named constants with no variables and no method bodies.

Interfaces

Interface: a collection of constants and *method declarations. An interface can't be instantiated.*

Methods in an interface do not have any code within statement body. Has a ';' after method definition line (signature).

speak and announce are method declarations in interface Speaker

```
public interface Speaker
{
      public void speak ();
      public void announce (String str);
}
```

Implementing Interfaces

Any class that extends Philosopher now is *subtype* of Speaker.

Philosopher class *must* declare a method for each of the methods declared in the interface

```
public class Philosopher implements Speaker
{
      public Philosopher (String thoughts)
      {
          philosophy = thoughts;
      }
      public void speak()
      {
          System.out.println(philosophy);
      }
      public void announce(String announcement)
      {
          System.out.println(announcement);
      }
      private String philosophy;
}
```

Interfaces

The Philosopher class could implement other methods for which there is no declaration in the Speaker interface, but it must have implementations of each interface method.

A class *implements* an interface by providing method implementations for each method defined in the interface. The implementing class is a subtype of the interface.

The keyword *implements* in the Philosopher class header says the class is defining bodies for each method in the interface.

Another Interface Example

roll and value are method declarations in interface Rollable

```
public interface Rollable
{
      // Reselect the upward-pointing face of object
      public void roll();
      // return the current value of the object
      public int value();
}
```

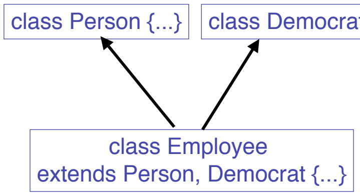
This interface specifies that there must be roll and value methods in each object that implements it.

```
public class Die implements Rollable {...}
```

Now Die is a subtype of the Rollable type. we can use a Die object anywhere Rollable objects are required.

Multiple Inheritance

The ability to derive a class from more than one parent class is known as *multiple inheritance.*



class Person {...}

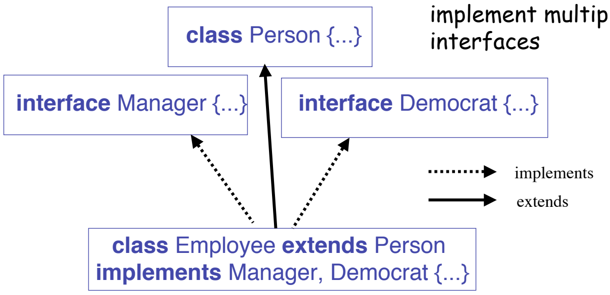
class Democrat {...}

class Employee
extends Person, Democrat {...}

Multiple inheritance is NOT ALLOWED in Java (i.e., a class can't extend more than one other class)

A Java Provision for Multiple Inheritance...

Classes can implement multiple interfaces



class Person {...}

interface Manager {...}

interface Democrat {...}

implements

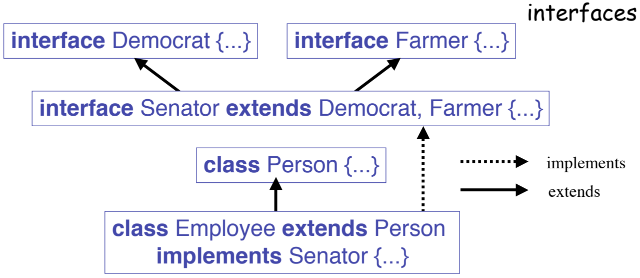
extends

class Employee **extends** Person
implements Manager, Democrat {...}

The Employee class would be a subclass of Person and a subtype of Manager and Democrat. We could write a program that makes use of Employee objects anywhere Person, Manager, or Democrat objects are required!

Another Way Java Provides for Multiple Inheritance...

Interfaces can extend multiple interfaces



interface Democrat {...}

interface Farmer {...}

interface Senator **extends** Democrat, Farmer {...}

class Person {...}

implements

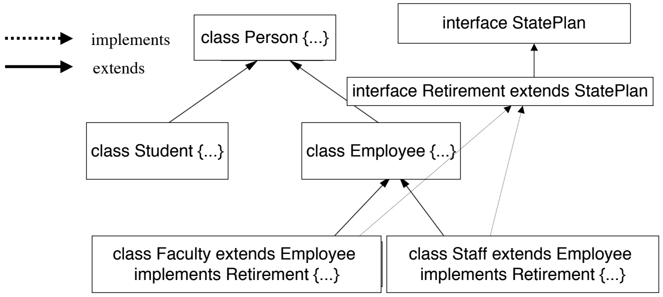
extends

class Employee **extends** Person
implements Senator {...}

The Employee class would be a subclass of Person and a subtype of Senator. We could write a program that makes use of Employee objects anywhere Person, Senator, Farmer, or Democrat objects are required!

Multiple Interfaces

When a class implements an interface that extends another interface, it must include all methods from *each* interface in hierarchy



implements

extends

class Person {...}

interface StatePlan

interface Retirement extends StatePlan

class Student {...}

class Employee {...}

class Faculty extends Employee
implements Retirement {...}

class Staff extends Employee
implements Retirement {...}

Faculty and Staff must include methods of both Retirement and StatePlan interfaces

Dynamic Binding and Polymorphism

Even though you can't create an object from an interface, you can use the interface as a type when you declare variables. The following code is legal:

```
StatePlan s = new Faculty ( “Joe”, 26, “1 Main St”,
                        “845-555-1212”, 10000.0 );
StatePlan e = new Staff ( “Boss”, 42, “4 Main St”,
                        “854-555-1212”, 10000.0 );
StatePlan p = null;
if( ((int)(Math.random() * 10)) % 2 == 1 )
      p = s;
else
      p = e;
System.out.println( “Person is “ + p );
```

Do not know until program runs whether to use `Faculty's toString` or `Staff's toString`

Type Compatibility

- Because a `Student` IS-A `Person`, a `Student` object can be reference by a `Person` type variable

```
Student s = new Student ( “Joe”, 26, “1 Main St”,
                          “845-555-1212”, 4.0 );
Person p = s;
System.out.println( “age is ” + p.getAge() );
```

- `p` may reference any object that IS-A `Person`
- Any method defined in the `Person` class or defined in the `Person` class and overridden in the `Student` class can be invoked through the `p` reference

Type Compatibility

```
Student s = new Student ( “Joe”, 26, “1 Main St”,
                          “845-555-1212”, 4.0 );
Person p = s;
System.out.println( “age is ” + p.getAge() );
System.out.println( “GPA is ” + p.getGPA() );
      // LINE ABOVE CAUSES ERROR
```

- But we can’t call methods defined *only* in class `Student` by using the reference p as it appears above. This is because a `Person` is not necessarily a `Student.`

Type Compatibility

```
Student s = new Student ( “Joe”, 26, “1 Main St”,
                          “845-555-1212”, 4.0 );
Person p = s;
System.out.println( “age is ” + p.getAge() );
System.out.println( “GPA is ” +
                         ((Student)p).getGPA() );
// LINE ABOVE IS OK if we cast p as a Student
```

- if p is cast as a `Student` the code works
- RULE: If a superclass identifier references a subclass object, then you need to cast the identifier using (subclass) cast when calling a subclass method.

Abstract Classes

- Abstract classes lie between interfaces and complete classes.

  ==> Class that may contain empty method declarations as well as fully defined methods and instance variables.

  - ➢ Not possible to instantiate an abstract class.
  - ➢ Subclasses must provide an implementation for each abstract method in the parent class.
  - ➢ "Partial" implementation of a class. Derived classes complete the definition.

```
abstract public class Matrix implements Graph {...}
```

An Abstract Class

The purpose of an abstract class is to define inheritable, shared variables and methods and to impose requirements through abstract methods.

```
Public abstract class Attraction {
   public int minutes;
   public Attraction() {minutes = 75;}
   public Attraction(int m) {minutes = m;}
   public int getMinutes() {return minutes;}
   public void setMinutes(int m) {minutes = m;}
   public abstract int rating();
}
```

Any classes derived from Attraction would inherit the public members and would have to provide an implementation of the abstract method rating.

A Class Derived from Attraction

```
public class Movie extends Attraction {
   public int script, acting, direction;

   public Movie() {script=5; acting=5; direction = 5;}

   public Movie(int m) {super(m);}

   public int rating() {
      return script+acting+direction+getMinutes();
}
```

Any classes derived from Attraction would inherit the public members and would have to provide an implementation of the abstract method rating.

```
class GenericArray {
  public static void main (String[] args) {
    Object[] array = new Object[4];
    array[0] = "String 1";
    array[1] = new Integer(1);
    array[2] = new Person();
    array[3] = new Integer("57");
    for (int i = 0; i < array.length; i++) {
      if (array[i] instanceof String) {
        String temp = (String)array[i];
        System.out.println(temp);
      }
      else if (array[i] instanceof Integer) {
        int x = ((Integer)array[i]).intValue();
        System.out.println(x);
      }
      else if (array[i] instanceof Person) {
        int y = ((Person)array[i]).getAge();
        System.out.println(y);
      }
    }
  }
}
```

Example of creating array of Objects and testing and casting each before printing

Reading Command-Line Arguments

- Command-line arguments are read through the main method's array of Strings parameter, args (or whatever you call it).
- Since command-line arguments are Strings, they must be converted to whatever types your program requires.
- Common to read the names of input and output files from the command-line.

Appendix

An example class and test routine. Try to figure out what it does, looking up the constructs you don’t understand

```
class Ticket
{
    public Ticket( )
    {
        System.out.println( "Calling "+
                            "constructor" );
        serialNumber = ++ticketCount;
    }

    public int getSerial( )
    {
        return serialNumber;
    }

    public String toString( )
    {
        return "Ticket #" + getSerial( );
    }

    public static int getTicketCount( )
    {
        return ticketCount;
    }

    private int serialNumber;
    private static int ticketCount = 0;
}
```

```
class TestTicket
{
    public static void main( String [ ]
                              args )
    {
        Ticket t1;
        Ticket t2;

        System.out.println( "Ticket count"
              +" is " +
              Ticket.getTicketCount( ) );

        t1 = new Ticket( );
        t2 = new Ticket( );

        System.out.println( "Ticket count"
              +" is " +
              Ticket.getTicketCount( ) );

        System.out.println(
                      t1.getSerial( ) );
        System.out.println(
                      t2.getSerial( ) );
    }
}
```

Another Example of Inheritance

```
public class Thought {

  //prints a message

  public void message() {

    System.out.println("I feel like I'm diagonally parked in "+
                       "a parallel universe.");

    System.out.println();
  }
}
```

Another Example of Inheritance

```
public class Advice extends Thought {

  // prints a message by overriding parent's version.  Then
  // explicitly calls parent method using super

  public void message(){
      System.out.println("Warning:  Dates in calendar are "+
                         "closer than they appear.");

      System.out.println();

      super.message();
  }
}
```

Another Example of Inheritance

```
public class Messages {
  // instantiates 2 objects and invokes the message
  // method in each

  public static void main(String[] args) {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message(); //overridden
  }
}
```

Another Example of Inheritance

```
public interface Transportable
{
    public static final int MAXINT = 1783479;
    public int weight();
    public boolean isHazardous();
}
```

```
public interface Sellable
{
    public String description();

    public int listPrice();

    public int lowestPrice();

}
```

```
public interface InsurableItem extends Transportable,
                                            Sellable
{
  public int insuredValue();
}
```

```
public class Photograph implements Sellable {
  private String descript;
  private int price;
  private boolean color;

  public Photograph(String desc, int p, boolean c) {
    descript = desc;
    price = p;
    color = c;
  }
  public String description() { return descript;}
  public int listPrice() {return price;}
  public int lowestPrice() {return price/2;}
}
```

```
public class BoxedItem implements InsurableItem {
  private String descript;
  private int price = MAXINT,weight,height=0,width=0,depth=0;
  private boolean haz;

    public BoxedItem(String desc, int p, int w, boolean h) {
      descript = desc;
      price = p;
      weight = w;
      haz = h;
    }
    public String description() {return descript;}
    public int listPrice() {return price;}
    public int lowestPrice() {return price/2;}
    public int weight() {return weight;};
    public boolean isHazardous() {return haz;}
    public int insuredValue() {return price*2;}

    public boolean equals (Sellable x){
       if (x instanceof BoxedItem){
         return x.listPrice()== this.price && x.weight() ==
                                   this.weight;
       }
       return false;
    }
}
```

```
class TestSellable {
  public static void main(String[] args) {
    Photograph p = new Photograph("landscape", 5000, true);
    BoxedItem b = new BoxedItem("statue", 10000, 2000,false);
    BoxedItem c = new BoxedItem("rug", 2000, 50, true);
    BoxedItem a = new BoxedItem("statue", 10000, 2000,false);
    InsurableItem s = null;

    if (b.equals(p))
      System.out.println("b and p equal");
    else System.out.println("b not equal to p");
    if (b.equals(c))
      System.out.println("b and c equal");
    else System.out.println("b not equal to c");
    if (b.equals(a)){
      s = a;
      System.out.println("b, s, and a equal");
    }
    else System.out.println("b not equal to a");

  }
}
```