# Class Relationships

- **Classes in a software system can have various types of relationships to each other**
- **Three of the most common relationships:**
  - **Dependency: A *uses* B**
  - **Aggregation: A *has-a* B**
  - **Inheritance: A *is-a* B**
- **Let's discuss dependency and aggregation further**
- **Inheritance later**

# Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

# Dependency

- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the `concat` method of the `String` class takes as a parameter another `String` object

```
str3 = str1.concat(str2);
```

- This drives home the idea that the service is being requested from a particular object

# Dependency

- The following example defines a class called `Rational` to represent a rational number
- A rational number is a value that can be represented as the ratio of two integers
- Some methods of the `Rational` class accept another `Rational` object as a parameter

# Using Rational Class

```
RationalNumber r1 = new RationalNumber (6, 8);
 RationalNumber r2 = new RationalNumber (1, 3);
 RationalNumber r3, r4, r5, r6, r7;

 System.out.println ("First rational number: " + r1);
 System.out.println ("Second rational number: " + r2);

 if (r1.equals(r2))
    System.out.println ("r1 and r2 are equal.");
 else
    System.out.println ("r1 and r2 are NOT equal.");

 r3 = r1.reciprocal();
 System.out.println ("The reciprocal of r1 is: " + r3);

 r4 = r1.add(r2);
 r5 = r1.subtract(r2);
 r6 = r1.multiply(r2);
 r7 = r1.divide(r2);

 System.out.println ("r1 + r2: " + r4);
 System.out.println ("r1 - r2: " + r5);
 System.out.println ("r1 * r2: " + r6);
 System.out.println ("r1 / r2: " + r7);
```

```
public class RationalNumber {
  private int numerator, denominator;

  //-----------------------------------------------------------------
  //  Constructor: Sets up the rational number by ensuring a nonzero
  //  denominator and making only the numerator signed.
  //-----------------------------------------------------------------
  public RationalNumber (int numer, int denom)   {
    if (denom == 0)
          denom = 1;

    // Make the numerator "store" the sign
    if (denom < 0)    {
          numer = numer * -1;
          denom = denom * -1;
    }

    numerator = numer;
    denominator = denom;

    reduce();
  }
```

```
//-----------------------------------------------------------------
  //  Returns the numerator of this rational number.
  //-----------------------------------------------------------------
  public int getNumerator ()
  {
     return numerator;
  }

  //-----------------------------------------------------------------
  //  Returns the denominator of this rational number.
  //-----------------------------------------------------------------
  public int getDenominator ()
  {
     return denominator;
  }

  //-----------------------------------------------------------------
  //  Returns the reciprocal of this rational number.
  //-----------------------------------------------------------------
  public RationalNumber reciprocal ()
  {
     return new RationalNumber (denominator, numerator);
  }
```

```
  //  Adds this rational number to the one passed as a parameter.
    //  A common denominator is found by multiplying the individual
    //  denominators.
    //-----------------------------------------------------------------
    public RationalNumber add (RationalNumber op2)   {
              int commonDenominator = denominator * op2.getDenominator();
              int numerator1 = numerator * op2.getDenominator();
              int numerator2 = op2.getNumerator() * denominator;
              int sum = numerator1 + numerator2;

              return new RationalNumber (sum, commonDenominator);
    }

public RationalNumber subtract (RationalNumber op2)   {
              int commonDenominator = denominator * op2.getDenominator();
              int numerator1 = numerator * op2.getDenominator();
              int numerator2 = op2.getNumerator() * denominator;
              int difference = numerator1 - numerator2;

              return new RationalNumber (difference, commonDenominator);
    }
```

```
//-----------------------------------------------------------------
  //  Multiplies this rational number by the one passed as a
  //  parameter.
  //-----------------------------------------------------------------
  public RationalNumber multiply (RationalNumber op2)
  {
     int numer = numerator * op2.getNumerator();
     int denom = denominator * op2.getDenominator();

     return new RationalNumber (numer, denom);
  }

  //-----------------------------------------------------------------
  //  Divides this rational number by the one passed as a parameter
  //  by multiplying by the reciprocal of the second rational.
  //-----------------------------------------------------------------
  public RationalNumber divide (RationalNumber op2)
  {
     return multiply (op2.reciprocal());
  }
```

```
public boolean equals (RationalNumber op2)
 {
   return ( numerator == op2.getNumerator() &&
            denominator == op2.getDenominator() );
 }

 //-----------------------------------------------------------------
 //  Returns this rational number as a string.
 //-----------------------------------------------------------------
 public String toString ()
 {
   String result;

   if (numerator == 0)
     result = "0";
   else
     if (denominator == 1)
       result = numerator + "";
     else
       result = numerator + "/" + denominator;

   return result;
 }
```

# Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A car *has a* chassis
- In software, an aggregate object contains references to other objects as instance data
- The aggregate object is defined in part by the objects that make it up
- This is a special kind of dependency – the aggregate usually relies on the objects that compose it

# Aggregation

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end

# StudentBody.java

```
Address school = new Address ("800 Lancaster Ave.", "Villanova",
                              "PA", 19085);

Address jHome = new Address ("21 Jump Street", "Lynchburg",
                             "VA", 24551);
Student john = new Student ("John", "Smith", jHome, school);


Address mHome = new Address ("123 Main Street", "Euclid", "OH",
                             44132);
Student marsha = new Student ("Marsha", "Jones", mHome, school);

System.out.println (john);
System.out.println ();
System.out.println (marsha);
```

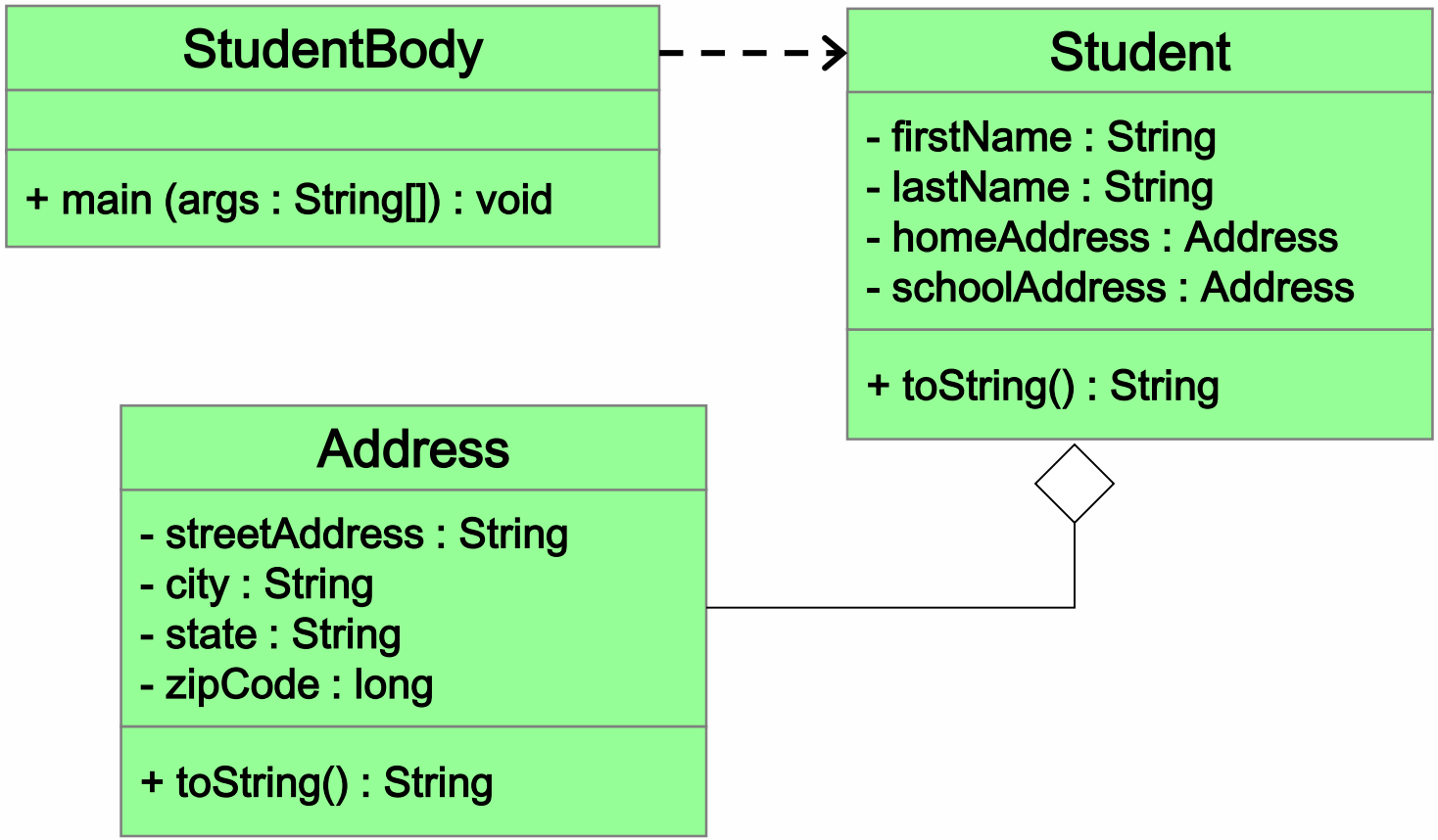
# Student.java

```
public class Student
{
   private String firstName, lastName;
   private Address homeAddress, schoolAddress;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this student with the specified values.
   //-----------------------------------------------------------------
   public Student (String first, String last, Address home, Address school)   {
      firstName = first;
      lastName = last;
      homeAddress = home;
      schoolAddress = school;
   }

public String toString()
   { ….}
}
```

# Address

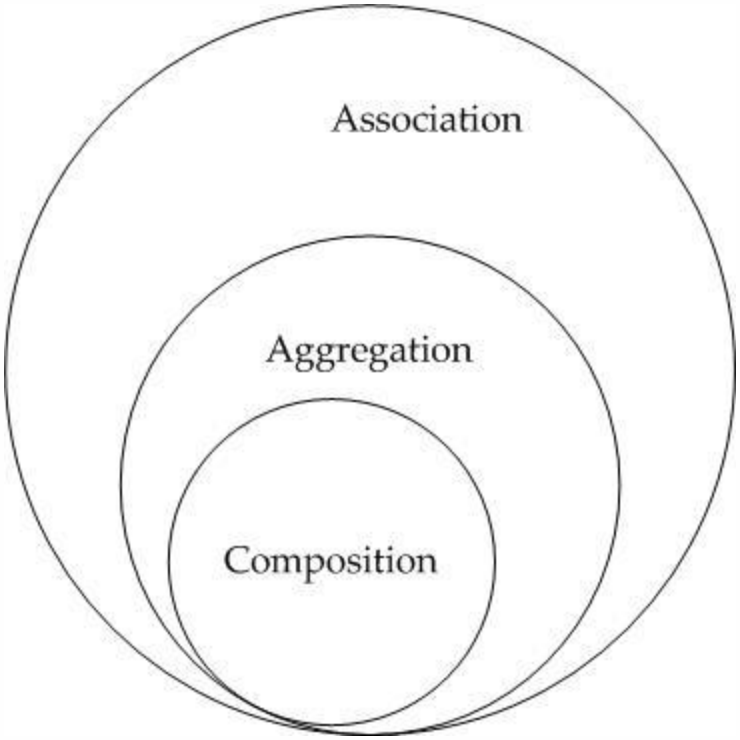
```
public class Address {
   private String streetAddress, city, state;
   private long zipCode;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this address with the specified data.
   //-----------------------------------------------------------------
   public Address (String street, String town, String st, long zip)   {
      streetAddress = street;
      city = town;
      state = st;
      zipCode = zip;
   }

   //-----------------------------------------------------------------
   //  Returns a description of this Address object.
   //-----------------------------------------------------------------
   public String toString() {}
}
```

# Aggregation in UML



# Association, Aggregation, and Composition

- **Association - I have a relationship with an object. Foo uses Bar**
- **Aggregation - I have an object which I've borrowed from someone else. When Foo dies, Bar may live on.**
- **Composition - I own an object and I am responsible for its lifetime, when Foo dies, so does Bar**

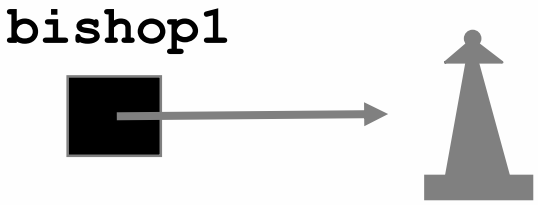


# References

- Recall that an object reference holds the memory address of an object

- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object

```
ChessPiece bishop1 = new ChessPiece();
```



# References

- Things you can do with a reference:
  - Declare it : String st;
  - Assign a new value to it
    - st = new String(“java”);
    - st = st2;
    - st = null;
  - Interact with the object using “dot” operator : st.length()
  - Check for equivalence
    - (st == st2)
    - (st == null)

# The null Reference

- An object reference variable that does not currently point to an object is called a *null reference*
- The reserved word `null` can be used to explicitly set a null reference:

```
name = null;
```

or to check to see if a reference is currently null:

```
if (name == null)
    System.out.println ("Invalid");
```

# The null Reference

- An object reference variable declared at the class level (an instance variable) is automatically initialized to null
- The programmer must carefully ensure that an object reference variable refers to a valid object before it is used
- Attempting to follow a null reference causes a `NullPointerException` to be thrown
- Usually a compiler will check to see if a local variable is being used without being initialized

# Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the argument (the value passed) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the argument and the formal parameter become aliases of each other

# Passing Objects to Methods

- **What a method does with a parameter may or may not have a permanent effect (outside the method)**
- **Note the difference between changing the internal state of an object versus changing which object a reference points to**

# ParameterPassing

```
ParameterTester tester = new ParameterTester();

    int a1 = 111;
    Num a2 = new Num (222);
    Num a3 = new Num (333);

    System.out.println ("Before calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

    tester.changeValues (a1, a2, a3);

    System.out.println ("After calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
```

# ParameterTester

```
class ParameterTester
{
  //-----------------------------------------------------------------
  //  Modifies the parameters, printing their values before and
  //  after making the changes.
  //-----------------------------------------------------------------
  public void changeValues (int f1, Num f2, Num f3)
  {
    System.out.println ("Before changing the values:");
    System.out.println ("f1\tf2\tf3");
    System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

    f1 = 999;
    f2.setValue(888);
    f3 = new Num (777);

    System.out.println ("After changing the values:");
    System.out.println ("f1\tf2\tf3");
    System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
  }
}
```

# Num

```
class Num {
   private int value;

public Num (int update) {
      value = update;
   }

public void setValue (int update)
   {
      value = update;
   }

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

# Method Overloading

- ***Method overloading*** **is the process of giving a single method name multiple definitions**
- **If a method is overloaded, the method name is not sufficient to determine which method is being called**
- **The *signature* of each overloaded method must be unique**
- **The signature includes the number, type, and order of the parameters**

# Method Overloading

- **The compiler determines which method is being invoked by analyzing the parameters**

```
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
   return x*y;
}
```

**Invocation**

```
result = tryMe(25, 4.32)
```

# Method Overloading

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

# Overloading Methods

- **The return type of the method is not part of the signature**
- **That is, overloaded methods cannot differ only by their return type**
- **Constructors can be overloaded**
- **Overloaded constructors provide multiple ways to initialize a new object**

# Overloading Methods

- **Constructors can be overloaded**
- **An overloaded constructor provides multiple ways to set up a new object**

# Snake Eyes

```
final int ROLLS = 500;
int snakeEyes = 0, num1, num2;

Die die1 = new Die();    // creates a six-sided die
Die die2 = new Die(20);  // creates a twenty-sided die

for (int roll = 1; roll <= ROLLS; roll++)
{
   num1 = die1.roll();
   num2 = die2.roll();

   if (num1 == 1 && num2 == 1)  // check for snake eyes
      snakeEyes++;
}

System.out.println ("Number of rolls: " + ROLLS);
System.out.println ("Number of snake eyes: " + snakeEyes);
System.out.println ("Ratio: " + (float)snakeEyes/ROLLS);
```

# Die Class

```
public class Die {
   private final int MIN_FACES = 4;
   private int numFaces;   // number of sides on the die
   private int faceValue;  // current value showing on the die

   //  Defaults to a six-sided die. Initial face value is 1.
   public Die () {
      numFaces = 6;
      faceValue = 1;
   }
   //  Explicitly sets the size of the die. Defaults to a size of
   //  six if the parameter is invalid.  Initial face value is 1.
   public Die (int faces)  {
      if (faces < MIN_FACES)
         numFaces = 6;
      else
         numFaces = faces;
      faceValue = 1;
   }
```

# Die Cont.

```
  //-----------------------------------------------------------------
    //  Rolls the die and returns the result.
    //-----------------------------------------------------------------
    public int roll ()
    {
       faceValue = (int) (Math.random() * numFaces) + 1;
       return faceValue;
    }

    //-----------------------------------------------------------------
    //  Returns the current die value.
    //-----------------------------------------------------------------
    public int getFaceValue ()
    {
       return faceValue;
    }
}
```