Principled Programming
Introduction to Coding in Any Imperative Language

Tim Teitelbaum
Emeritus Professor
Department of Computer Science
Cornell University

Classes and Objects
A taxonomy is a system of classification. Taxonomies are an essential mechanism for organizing subject matter. Hierarchical taxonomies in which concepts are organized into tree structures are ubiquitous. In a hierarchy, the most general concept is placed at the root of the tree, and subordinate concepts branch out from there. Each category is a set of individuals. A Venn diagram depicts categories as nested regions, and individuals as dots.
Taxonomic categories in programming are called *classes*, and the individuals of a class are its *objects*. The root category is `Object`.

We illustrate classes and objects by implementing `Pair`, `Fraction`, and `Rational`. Every rational is a fraction, and every fraction is a pair of integers, and every pair is an `Object`.

We then implement `ArrayList<E>`, a parameterized class for representing and manipulating collections of type `E` elements. We use the class `ArrayList<Rational>` to complete code for enumerating rationals. Because `ArrayList<E>` is similar to the library class `HashSet<E>`, it is easy to replace one with the other, and compare their speed. We do so, and demonstrate the dramatic speedup of hash tables over lists.

Finally, in a bit of a double cross, we observe that collections weren’t actually needed for enumerating rationals, and obtain a still-faster implementation without them.
A class is a collection of variable declarations and method definitions. An object is a dynamic instantiation of the variables (and methods) of a class whose declarations (and definitions) are not prefixed by the modifier static.

Such variables are known as object fields or instance variables (and such methods are known as instance methods). Objects and references to them are depicted as shown.

Classes are types. If C is a class, a variable v of type C is obtained by executing the declaration:

```java
C v;
```

Such a variable can hold a reference to an object of type C.
An object $o$ of type $C$ is created by executing the expression

```
new C(…)
```

If object $o$ has a field $f$, the field is accessed as $o.f$.

If object $o$ has a method $m$, the method is invoked by $o.m(…)$.

If a class is a shape of cookie (with its fields and methods), and objects are the cookies themselves, then `new C(...)` is a cookie-cutter that stamps out new cookies (with instances of $C$’s instance fields and methods).

In contrast, a `static` variable (or a `static` method) is `unique`, and is not instantiated for each object. All objects of a class share access to such variables (and methods).
Class definition:

class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */

Variable declaration (with initialization):

Pair v = new Pair(2,3);
Class definition:

class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */

Variable declaration (with initialization):

Pair v = new Pair(2,3);
Class definition:

class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */

Variable declaration (with initialization):
Pair v = new Pair(2,3);
Class definition:

```java
class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */
```

Variable declaration (with initialization):

```java
Pair v = new Pair(2,3);
```
Class definition:

class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */

Variable declaration (with initialization):

Pair v = new Pair(2,3);
Visibility: Each field and method of a class has visibility public, private, or protected.

class Pair {
  protected int key;
  protected int value;
  /* Constructor. */
  public Pair(int k, int v) { key = k; value = v; }
  /* Access. */
  public int getKey()   { return key; }
  public int getValue() { return value; }
} /* Pair */

• public fields and methods are globally visible (the default).
• private fields and methods are only visible within the class.
• protected fields and methods are only visible within the class, or within a subclass of the class, e.g., Fraction.
Modifiability: A `private` or `protected` field with a `public` getter is read-only outside its scope.

class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */

- E.g., clients of Pair can obtain the components of a Pair using the getter, but cannot change those fields. Such an object is said to be `immutable`. 
Default String representation:

- Every Pair is an Object, and every Object has a default `toString` method.
- However, the String representation provided by that method is not particularly helpful.

Output the String representation of an object:

```java
System.out.println( v );
```
Overriding definition of toString for pairs:

class Pair {
    ...
    /* String representation of this. */
    public String toString() { return "<" + key + "," + value + ">"; } 
} /* Pair */

Output the String representation of an object:

System.out.println( v );

<2,3>
Overriding definition of toString for pairs:

class Pair {
    
    /* String representation of this. */
    public String toString() { return "<" + key + "," + value + ">"; }
} /* Pair */

Output the String representation of an object:

System.out.println( v );
Override definition of `toString`:

class Pair {
   ...
   /* String representation of this. */
   public String toString() { return "<" + key + "," + value + ">"; }
} /* Pair */

Output the String representation of an object:

System.out.println( v );
Overriding definition of toString:

class Pair {
   ...
   /* String representation of this. */
   public String toString() { return "<" + key + "," + value + ">"; }
} /* Pair */

Output the String representation of an object:
System.out.println(v);
Overriding definition of toString:

class Pair {
  ...
  /* String representation of this. */
  public String toString() { return "<" + key + "," + value + ">"; }
} /* Pair */

Output the String representation of an object:

System.out.println( v );
The definition of operator == for objects is identity.

- “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

```java
Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);  // false
System.out.println(z2==z3);  // true
```
Demonstrate the difference between identity and equality.

Object
  Pair
  Fraction
  Rational

The definition of operator == for objects is identity.

- “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);
System.out.println(z2==z3);
Demonstrate the difference between identity and equality.

Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);
System.out.println(z2==z3);

• “Identity” means “exactly the same object”.

The definition of operator == for objects is identity.
Demonstrate the difference between identity and equality.

Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);
System.out.println(z2==z3);

The definition of operator == for objects is identity.

• “Identity” means “exactly the same object”.
Demonstrate the difference between identity and equality.

Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);  // false
System.out.println(z1==z3);

The definition of operator == for objects is identity.

• “Identity” means “exactly the same object”.

• The definition of operator == for objects is identity.
The definition of operator == for objects is identity.

• “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2); // false
System.out.println(z2==z3); // true
Demonstrate the difference between identity and equality.

Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1.equals(z2)); // false
System.out.println(z2.equals(z3)); // true

The default definition of equals for Object values is also identity.

• “Identity” means “exactly the same object”.
• Every Object has an equals method that can be applied to another Object to test “equality”, which is user-definable.
• The default definition of method equals in Object is identity, i.e., the same as ==.
The default definition of equals can be overridden.

- “Identity” means “exactly the same object”.
- Every Object has an equals method that can be applied to another Object to test “equality”, which is user-definable.
- The default definition of method equals in Object is identity, i.e., the same as ==.
- Unlike the == operator, equals can be overridden, e.g., to treat non-identical pairs with equal components as equal.

Demonstrate the difference between identity and equality.

```java
Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1.equals(z2));  // true
System.out.println(z2.equals(z3));  // true
```

with the overriding definition of equals shown on the next slide
Overriding definition of equals for pairs.

class Pair {
    ...
    /* Equality. */
    @Override
    public boolean equals(Object q) {
        if (q == null) return false;
        if (q == this) return true;
        if (! (q instanceof Pair)) return false;
        Pair qPair = (Pair)q;
        return (key == qPair.key) &&
               (value == qPair.value);
    } /* equals */
} /* Pair */

Asks the compiler to warn if the next method definition is not overriding.
Overriding definition of `equals` for pairs.

class Pair {
    ...
    /* Equality. */
    @Override
    public boolean equals(Object q) {
        if (q==null) return false;
        if (q==this) return true;
        if ( !(q instanceof Pair) ) return false;
        Pair qPair = (Pair)q;
        return (key   == qPair.key) &&
            (value == qPair.value);
    } /* equals */
} /* Pair */

An Object is never equal to no Object.
Overriding definition of equals for pairs.

class Pair {
    ... 
    /* Equality. */
    @Override
    public boolean equals(Object q) {
        if (q==null) return false;
        if (q==this) return true;
        if ( !(q instanceof Pair) ) return false;
        Pair qPair = (Pair)q;
        return (key   == qPair.key) &&
               (value == qPair.value);
    } /* equals */
} /* Pair */

An Object is always equal to itself, e.g. z2 and z3.
Overriding definition of equals for pairs.

class Pair {
    ...
    /* Equality. */
    @Override
    public boolean equals(Object q) {
        if (q==null) return false;
        if (q==this) return true;
        if ( !(q instanceof Pair) ) return false;
        Pair qPair = (Pair)q;
        return (key == qPair.key) && (value == qPair.value);
    } /* equals */
    } /* Pair */

A Pair can only equal another Pair.

Demonstrate the difference between identity and equality.
Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1.equals(z2));   true
System.out.println(z2.equals(z3));   true
Overriding definition of equals for pairs.

class Pair {
    
    /* Equality. */
    @Override
    public boolean equals(Object q) {
        if (q==null) return false;
        if (q==this) return true;
        if ( !(q instanceof Pair) ) return false;
        Pair qPair = (Pair)q;
        return (key == qPair.key) &&
                (value == qPair.value);
    } /* equals */
} /* Pair */

A Pair can only equal another Pair, and then only when their components are equal, e.g. z1 and z2.
Subclass definition: Fraction

```java
class Fraction extends Pair {
    /* Constructor. */
    public Fraction(int numerator, int denominator) {
        super(numerator, denominator); // Apply the Pair constructor.
        assert denominator!=0: "0 denominator";
    }
    /* Access. */
    public int getNumerator() { return key; }
    public int getDenominator() { return value; }
} /* Fraction */
```

Getters have direct access to the fields key and value because they are declared protected in Pair, a superclass of Fraction.
Overriding definition of toString for fractions:

class Fraction extends Pair {
    /* Constructor. */
    public Fraction(int numerator, int denominator) {
        super(numerator, denominator); // Apply the Pair constructor.
        assert denominator!=0: "0 denominator";
    }
    /* Access. */
    public int getNumerator() { return key; }
    public int getDenominator() { return value; }
    /* String representation of this. */
    public String toString() { return key + "/" + value; }
} /* Fraction */
Overriding definition of equals for fractions:

- Not needed because two fractions are equal iff they have equal numerators and equal denominators.
**Subclass definition:** Rational

```java
class Rational extends Fraction {
    /* Constructor */
    public Rational(int numerator, int denominator) {
        super(numerator, denominator);  // Apply the Fraction constructor.
        int g = gcd(numerator, denominator); // Apply the Fraction constructor.
        key = numerator/g;
        value = denominator/g;
    }
    ...
} /* Rational */
```

Method gcd not shown.

Confirm that the denominator is not zero (checked by Function), and then update the representation to reduced form.
### Overriding definition of toString for rationals:

```java
class Rational extends Fraction {
    /* Constructor */
    public Rational(int numerator, int denominator) {
        super(numerator, denominator);   // Apply the Fraction constructor.
        int g = gcd(numerator, denominator);
        key = numerator/g;
        value = denominator/g;
    }

    /* String representation of this. */
    public String toString() {
        if ( value==1 ) return key + ""; // this as int
        else return super.toString(); // this as Fraction
    }
}
```
Overriding definition of `equals` for rationals not needed:

- Not needed because two rationals are equal iff they are equal as reduced fractions.
**Subtype polymorphism:** A variable of class C can be assigned a reference to any object of class C', where C' is either C itself, or C' is a subclass of C, i.e., lower in the class hierarchy.

Object o;
Subtype polymorphism:

```java
Object o;
o = new Pair(4,6);
```
Subtype polymorphism:

Object o;
o = new Pair(4,6);
o = new Fraction(4,6);
Subtype polymorphism:

Object o;
o = new Pair(4,6);
o = new Fraction(4,6);
o = new Rational(4,6);
Object o;
o = new Pair(4,6);
o = new Fraction(4,6);
o = new Rational(4,6);
o = new Rational(6,3);
Dynamic method dispatch: The definition used for any given method invocation depends on the type of the value, not the type of the variable that contains that value.

Object o;
Dynamic method dispatch:

Object o;
o = new Pair(4,6);   System.out.println( o );
Dynamic method dispatch:

```java
Object o;
o = new Pair(4,6); System.out.println( o );
o = new Fraction(4,6); System.out.println( o );
```

```
<4,6>
4/6
```
Dynamic method dispatch:

Object o;
o = new Pair(4,6); System.out.println( o );
o = new Fraction(4,6); System.out.println( o );
o = new Rational(4,6); System.out.println( o );
Object o;
o = new Pair(4,6); System.out.println( o );
o = new Fraction(4,6); System.out.println( o );
o = new Rational(4,6); System.out.println( o );
o = new Rational(6,3); System.out.println( o );
Subtype polymorphism caveat:
If variable \( v \) has type \( C \), a field access \( v.f \), or a method invocation \( v.m(...) \), requires that field \( f \) or method \( m \) necessarily exist in any object of type \( C \).

Object  \( o = \text{new } \text{Pair}(4,6); \)  System.out.println(\( o . \text{getKey}() \));  \( // \) Illegal.
Pair  \( p = \text{new } \text{Pair}(4,6); \)  System.out.println(\( p . \text{getKey}() \));  \( // \) Legal.
\( p = \text{new } \text{Pair}(2,3); \)  System.out.println(\( p . \text{getNumerator}() \));  \( // \) Illegal.
Fraction  \( r = \text{new } \text{Fraction}(4,6); \)  System.out.println(\( r . \text{getNumerator}() \));  \( // \) Legal.
Rational  \( q = \text{new } \text{Rational}(4,6); \)  System.out.println(\( q . \text{getNumerator}() \));  \( // \) Legal.
Inheritance: The class hierarchy is also called the *inheritance hierarchy*. Objects of class C are said to inherit all fields $f$ of superclasses of C above it in the hierarchy. They also inherit the most specific (overriding) version of method $m$ defined either in class C, or in one of C’s superclasses, i.e., the first definition of $m$ found in a traversal from C up to Object in the hierarchy.
Class definition:

class ArrayList {
    private int A[];  // ArrayList elements are in A[0..size-1].
    private int size;  // The default value is 0.

    ...
}
Array definition:

class ArrayList {
    private int A[];  // ArrayList elements are in A[0..size-1].
    private int size;  // The default value is 0.

    ...
Class definition:

class ArrayList {
    private int A[];  // ArrayList elements are in A[0..size-1].
    private int size;  // The default value is 0.

    /* Constructors. */
    public ArrayList( int m ) {
        if ( m<0 ) throw new IllegalArgumentException();
        A = new int[m];
    }

    public ArrayList() { this( 20 /* DEFAULT_SIZE */ );

    ...
}
A getter for the read-only field `size`, and a predicate to test for an empty list.

Class definition:

```java
class ArrayList {
    private int A[];  // ArrayList elements are in A[0..size-1].
    private int size;  // The default value is 0.

    /* Constructors. */
    public ArrayList( int m ) {
        if ( m<0 ) throw new IllegalArgumentException();
        A = new int[m];
    }

    public ArrayList() { this( 20 /* DEFAULT_SIZE */ );

    /* Size. */
    public int size() { return size; }
    public boolean isEmpty() { return size==0; }

    ...
```
Get and set the k-th element of the list. Set returns the old k-th value of the list.

```java
/* Access. */
public int get(int k) {
    checkBoundExclusive(k);
    return A[k];
}
public int set(int k, int v) {
    checkBoundExclusive(k);
    int old = A[k];
    A[k] = v;
    return old;
}
...
/* Access. */
public int get(int k) {
    checkBoundExclusive(k);
    return A[k];
}

public int set(int k, int v) {
    checkBoundExclusive(k);
    int old = A[k];
    A[k] = v;
    return old;
}

/* Utility */
private void checkBoundExclusive(int k) {
    if (k>=size) throw new IndexOutOfBoundsException("≥size");
}

private void checkBoundInclusive(int k) {
    if (k>size) throw new IndexOutOfBoundsException(">size");
}
/* Insertion / Deletion. */

public void add(int v) {
    if ( size == A.length )
        ensureCapacity( size+1 );
    A[size] = v; size++;
}

public void add(int k, int v) {
    checkBoundInclusive(k);
    if ( size == A.length )
        ensureCapacity( size+1 );
    for (int j=size; j>k; j--) A[j] = A[j-1];
    A[k] = v;
    size++;
}

/* Insertion / Deletion. */
Remove an element from list at end or at position k.

```java
/* Insertion / Deletion. */
public void add(int v) {
    if ( size==A.length ) ensureCapacity( size+1 );
    A[size] = v; size++;
}

public void add(int k, int v) {
    checkBoundInclusive(k);
    if ( size==A.length ) ensureCapacity( size+1 );
    for (int j=size; j>k; j--) A[j] = A[j-1];
    A[k] = v;
    size++;
}

public int remove(int k) {
    checkBoundExclusive(k);
    int old = A[k];
    size--;
    for (int j=k; j<size; j++) A[j] = A[j+1];
    return old;
}
...
/* Capacity. */

public void ensureCapacity( int minCapacity) {
    int currentLength = A.length;
    if ( minCapacity > currentLength ) {
        int B[] = new int[Math.max(2*currentLength, minCapacity)];
        for (int k=0; k<size; k++) B[k] = A[k];
        A = B;
    }
}

...
/* Capacity. */

    public void ensureCapacity( int minCapacity ) {
        int currentLength = A.length;
        if ( minCapacity > currentLength ) {
            int B[] = new int[Math.max(2*currentLength, minCapacity)];
            for (int k=0; k<size; k++) B[k] = A[k];
            A = B;
        }
    }

/* Membership. */

    public int indexOf( int v ) {  
        int k = 0; while ( (k<n) && (v!=A[k]) ) k++;
        if ( k==n ) return -1; else return k;
    }

    public boolean contains( int v ) {
        return indexOf(v)!=-1;
    }  

} /* ArrayList */
/* Output reduced positive fractions, i.e., positive rationals. */
/* set reduced = { }; */
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        int g = gcd(r+1, c+1);
        /* rational z = ((r+1)/g, (c+1)/g); */
        if (/* z is not an element of reduced */ ) {
            System.out.println( /* z */ );
            /* reduced = reduced U {z}; */
        }
        r--;
    }
    d++;
}
Enumeration of rationals: We can adopt Rational as the type of rational z.

```java
/* Output reduced positive fractions, i.e., positive rationals. */
/* set reduced = { }; */
int d = 0;
while ( true ) {
    int r = d;
    for ( int c=0; c<=d; c++ ) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        Rational z = new Rational(r+1, c+1);

        if ( /* z is not an element of reduced */ ) {
            System.out.println( z );
            /* reduced = reduced U {z}; */
        }
        r--;
    }
    d++;
}
```
Similarly, we would like to adopt ArrayList as the type of the set reduced, but cannot do so because as currently written it is a collection of int items, not Rational items.

**Enumeration of rationals:** We can adopt Rational as the type of rational z.

```java
/* Output reduced positive fractions, i.e., positive rationals. */
/* set reduced = { }; */
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        Rational z = new Rational(r+1, c+1);
        if (/* z is not an element of reduced */) {
            System.out.println( z );
            /* reduced = reduced U {z}; */
        }
        r--;
    }
    d++;
}
```
Enumeration of rationals: We need an ArrayList of Rational items.

This could be done by:
• Cloning the ArrayList of int implementation, and adapting the clone to be a collection of Rational elements (ugh!), or
• Parameterizing ArrayList to be ArrayList<E>, a collection of elements of arbitrary object type E, and then instantiating it as ArrayList<Rational>, a collection of Rational elements (far better!).

A parametrized class definition is called a generic class.
Generic class definition:

class ArrayList\<E\> {
    private E A[];       // ArrayList elements are in A[0..size-1].
    private int size;    // The default value is 0.

    /* Constructors. */
    public ArrayList( int m ) {
        if ( m<0 ) throw new IllegalArgumentException();
        A = (E[]) new Object[m];
    }

    public ArrayList() { this( 20 /* DEFAULT_SIZE */ ); }

    /* Size. */
    public int size() { return size; }
    public boolean isEmpty() { return size==0; }

    ...

    An array of arbitrary objects is created, and is cast to the type of A.

    The type of an ArrayList element is parameterized as E.
/* Access. */
public E get(int k) {
    checkBoundExclusive(k);
    return A[k];
}

public E set(int k, E v) {
    checkBoundExclusive(k);
    E old = A[k];
    A[k] = v;
    return old;
}

/* Utility */
private void checkBoundExclusive( int k ) {
    if (k>=size) throw new IndexOutOfBoundsException( "≥size" );
}

private void checkBoundInclusive( int k ) {
    if (k>size) throw new IndexOutOfBoundsException( ">size" );
}

...
/* Insertion / Deletion. */

public void add(E v) {
    if ( size==A.length ) ensureCapacity( size+1 );
    A[size] = v; size++;
}

public void add(int k, E v) {
    checkBoundInclusive(k);
    if ( size==A.length ) ensureCapacity( size+1 );
    for (int j=size; j>k; j--) A[j] = A[j-1];
    A[k] = v;
    size++;
}

public E remove(int k) {
    checkBoundExclusive(k);
    E old = A[k];
    size--;
    for (int j=k; j<size; j++) A[j] = A[j+1];
    return old;
}

...
/* Capacity. */
public void ensureCapacity( int minCapacity) {
    int currentLength = A.length;
    if ( minCapacity > currentLength ) {
        E B[] = (E[]) new Object[Math.max(2*currentLength, minCapacity)];
        for (int k=0; k<size; k++) B[k] = A[k];
        A = B;
    }
}

/* Membership. */
public int indexOf(E v) {
    int k = 0; while ( (k<n) && !v.equals(A[k]) ) k++;
    if ( k==n ) return -1; else return k;
}

public boolean contains(E v) {
    return indexOf(v)!=-1;
}

} /* ArrayList */

Use the equals method of the element type rather than ==.

An array of arbitrary objects is created, and is cast to the type of B.
Enumeration of rationals: Returning to the incomplete code for enumerating rationals.

/* Output reduced positive fractions, i.e., positive rationals. */
/* set reduced = { }; */
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        Rational z = new Rational(r+1, c+1);
        if (/* z is not an element of reduced */ ) {
            System.out.println( z );
            /* reduced = reduced ∪ {z}; */
        }
        r--;
    }
    d++;
}
*/ Output reduced positive fractions, i.e., positive rationals. */
ArrayList<Rational> reduced = new ArrayList();
int d = 0;
while ( true ) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        Rational z = new Rational(r+1, c+1);
        if ( !reduced.contains(z) ) {
            System.out.println( z );
            reduced.add(z);
        }
        r--;
    }
    d++;
Enumeration of rationals: and obtain the correct output.

1
2
1/2
3
1/3
4
3/2
2/3
1/4
5
1/5
6
5/2
4/3
3/4
2/5
1/6
7
5/3
3/5
1/7
etc.

2/2 omitted

4/2, 3/3, and 2/4 omitted

6/2 omitted

4/4 omitted

2/6 omitted
Uniformity:

- In some languages, all values are uniformly objects of a class.
- In other languages, there is a distinction between primitive values and objects of a class.
- Primitive values, e.g., values of types `int`, `long`, `float`, `double`, `boolean`, and `char`, fit conveniently into variables of standard sizes.
- In contrast, objects are accessed via references. The object reference has a standard size, but the object itself doesn’t.
- In the interest of efficiency, but at the expense of complexity, Java offers two worlds, one in which values are primitive, and the other in which values are objects.
- Crossing back and forth between the two worlds is a bit complicated, and will be addressed next.
Class definition: Recall the definition of class Pair.

class Pair {
    protected int key;
    protected int value;
    /* Constructor. */
    public Pair(int k, int v) { key = k; value = v; }
    /* Access. */
    public int getKey() { return key; }
    public int getValue() { return value; }
} /* Pair */
**Generic class definition:** It too can be made generic so we can have pairs of any object types.

```java
class Pair<K, V> {
    protected K key;
    protected V value;
    /* Constructor. */
    public Pair(K k, V v) { key = k; value = v; }
    /* Access. */
    public K getKey() { return key; }
    public V getValue() { return value; }
    ...
} /* Pair<K, V> */
```
**Generic class definition:** It too can be made generic so we can have pairs of any object type.

class Pair<K, V> {
    ...
    /* Equality. */
    @Override
    public boolean equals(Object q) {
        if (q == null) return false;
        if (q == this) return true;
        if (!(q instanceof Pair)) return false;
        Pair qPair = (Pair)q;
        return key.equals(qPair.key) &&
               value.equals(qPair.value);
    } /* equals */
} /* Pair */

Uses the equals methods of the component types (which need not be the same) rather than ==.
Pairs of any object type. The generic class Pair<K, V> can be instantiated with any object types for K and V.

For example, each of the following is a valid declaration:

Pair<Fraction, Fraction> ff;
Pair<Fraction, Rational> fr;
Pair<Fraction, Object> fo;
Pair<Pair<Fraction, Fraction>, Pair<Rational, Rational>> ffrr;

but the following is not a valid declaration:

Pair<int, int> ii;

because int is a primitive type, not an object type.

We deal with this next.
Boxed values.

There is an object type for each primitive type. Values of those types are called boxed primitive values:

- Integer
- Long
- Float
- Double
- Boolean
- Char

Java attempts to box and unbox values fully automatically, but you need to know that it is going on.

For example, if we were to change Pair to be the generic class Pair<K, V>, the definition of class Function would no longer be correct, as explained next.
Subclass definition: Recall the definition of Fraction.

class Fraction extends Pair {
   /* Constructor. */
   public Fraction(int numerator, int denominator) {
      super(numerator, denominator); // Apply the Pair constructor.
      assert denominator!=0: "0 denominator";
   }
   /* Access. */
   public int getNumerator() { return key; }
   public int getDenominator() { return value; }
} /* Fraction */
**Subclass definition:** Since Pair is now a generic class, Fraction must instantiate it with component types.

class Fraction extends Pair<Integer,Integer> {
    /* Constructor. */
    public Fraction(int numerator, int denominator) {
        super(numerator, denominator); // Apply the Pair constructor.
        assert denominator!=0: "0 denominator";
    }
    /* Access. */
    public int getNumerator() { return key; }
    public int getDenominator() { return value; }
} /* Fraction */
**Subclass definition:** Auto-boxing and auto-unboxing occurs between `int` values and `Integer` values.

```java
class Fraction extends Pair<Integer, Integer> {
    /* Constructor. */
    public Fraction(int numerator, int denominator) {
        super(numerator, denominator); // Apply the Pair constructor.
        assert denominator!=0: "0 denominator";
    }
    /* Access. */
    public int getNumerator() { return key; }
    public int getDenominator() { return value; }
} /* Fraction */
```

Auto-boxing of `int` parameters `numerator` and `denominator` occurs when they are passed to the `Pair` constructor since it expects `Integer` arguments.
**Subclass definition:** Auto-boxing and unboxing occurs between \texttt{int} values and \texttt{Integer} values.

```java
class Fraction extends Pair<Integer, Integer> {
    /* Constructor. */
    public Fraction(int numerator, int denominator) {
        super(numerator, denominator); // Apply the Pair constructor.
        assert denominator!=0: "0 denominator";
    }
    /* Access. */
    public int getNumerator() { return key; }
    public int getDenominator() { return value; }
} /* Fraction */
```

Auto-unboxing of the \texttt{Integer} key and value fields occurs when they returned as the values of the getters.
Subclass definition: Similarly, recall the definition of Rational.

class Rational extends Fraction {
    /* Constructor */
    public Rational(int numerator, int denominator) {
        super(numerator, denominator); // Apply the Fraction constructor.
        int g = gcd(numerator, denominator);
        key = numerator/g;
        value = denominator/g;
    }
    ...
} /* Rational */
Subclass definition: Since Fraction inherits from Pair<Integer,Integer>, so too does Rational.

class Rational extends Fraction {
    /* Constructor */
    public Rational(int numerator, int denominator) {
        super(numerator, denominator);    // Apply the Fraction constructor.
        int g = gcd(numerator, denominator);
        key = numerator/g;
        value = denominator/g;
    }
    ...
} /* Rational */

No auto-boxing of int parameters numerator and denominator occurs when they are passed to the Function constructor because it expects two int arguments. They are auto-boxed when it invokes the Pair constructor.
Subclass definition: Since Fraction inherits from Pair<Integer, Integer>, so too does Rational.

class Rational extends Fraction {
    /* Constructor */
    public Rational(int numerator, int denominator) {
        super(numerator, denominator);    // Apply the Fraction constructor.
        int g = gcd(numerator, denominator);
        key = numerator/g;
        value = denominator/g;
    }
    ...
} /* Rational */

Auto-boxing of the computed int values numerator/g and denominator/g occurs when they are assigned to the Integer key and value fields.
Polymorphism: Four kinds have been illustrated.

• **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

• **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

• **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

• **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.
Polymorphism: Four kinds have been illustrated.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.

• e.g., the object constructed by Rational(2,3) can be treated as a Rational, Fraction, Pair, or Object.
Polymorphism: Four kinds have been illustrated.

• **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

• **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

• **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

• **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.

e.g., a variable declared to have type Fraction can be assigned a Fraction or Rational, but it cannot be assigned a Pair or Object.
Polymorphism: Four kinds have been illustrated.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.
Polymorphism: Four kinds have been illustrated.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.
Polymorphism: Four kinds have been illustrated.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.
Polymorphism: Four kinds have been illustrated.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.

---

e.g., the boxing of an `int` in the `Fraction` constructor, and the unboxing of an `Integer` in the `Rational` getters.
**Polymorphism:** Four kinds have been illustrated.

- **Subtype polymorphism,** where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism,** where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion,** where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading,** where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.

e.g., the cast `(E[])` in the statement `A = (E[]) new Object[m];` in the `ArrayList<E>` constructor.
Polymorphism: Four kinds have been illustrated.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.

- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).

- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is implicitly converted to the required type. Other conversions are explicit, e.g., casts. Another term for conversion is coercion.

- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation.

E.g., `ArrayList<E>` has two constructors, one with no parameter, and the other with one parameter. It also has two `add` methods, one with one parameter, and the other with two parameters.
Garbage Collection. An object dies when it can no longer be accessed in the program.

- Objects consume space in computer memory.
- Space consumed by objects that can no longer be accessed can be reclaimed automatically by a mechanism (that runs behind the scene) called *garbage collection*.
- Normally, you don’t have to think about such matters. However, you should be aware that retaining a gratuitous reference to an object can cause it to be needlessly retained.
- By itself, one such object is no big concern. But if it is at the beginning of a chain of references from one object to another, then that one gratuitous reference can be the cause of an unbounded number of needlessly-retained other objects, which is of concern.
- This is why we make sure that an `ArrayList<E>` retains no gratuitous references to objects in the unused suffix of the array.
- We explain how this works next. It is a bit subtle, but is instructive.
Garbage Collection. Recall the definition of `remove` in `ArrayList<E>`.

```java
public E remove(int k) {
    checkBoundExclusive(k);
    E old = A[k];
    size--;
    for (int j=k; j<size; j++) A[j] = A[j+1];
    return old;
}
```

The left shift of (blue) values overwrites the (green) value in `A[k]` that is to be removed from the collection. It was a reference to some object, and if this was the only reference to that object, it can be garbage collected. The value at `A[k]` being removed is not the issue.
Garbage Collection. Recall the definition of remove in `ArrayList<E>`.

```java
public E remove(int k) {
    checkBoundExclusive(k);
    E old = A[k];
    size--;
    for (int j=k; j<size; j++) A[j] = A[j+1];
    return old;
}
```

The last (blue) value in the collection, which was originally in `A[size-1]`, shifts left but because the shift is effected by *copying*, the original instance of that value (violet) would also remain as `A[size]`, the first element of the unused array suffix. It is this violet instance of the value that we nullify. Note that the object referred to by the violet reference cannot yet be collected because a reference to it remains in `A[size-1]`. However, if and when *that* reference is removed or is overwritten with a new value by `set`, the object in question will be collectable.
Libraries: Classes that you can learn and use.

Libraries are extensions of the core language. The standard library includes:

• **Object**, the root of the class inheritance hierarchy. All other classes are subclasses of **Object**, and inherit methods from it.
• **Math**, a class that contains built-in mathematical functions as **static** methods.
• **String**, the class for sequences of Unicode characters. String constants, e.g., "a String", are references to String objects that contain the given sequence of characters.
• **Integer**, **Boolean**, etc., classes for the boxed primitive values.
Libraries: The library `java.util` contains many useful classes, including these for collections:

```
Object
  \--------
|        |
|        |
|        |
AbstractCollection<E>
  \--------
|        |
|        |
|        |
AbstractList<E>   AbstractSet<E>   AbstractMap<K,V>
```

Class `ArrayList<E>`, which we (partially) implemented ourselves, appears in the inheritance hierarchy as a second cousin of `HashSet<E>`, a familial relationship that we would have obtained by writing:

```
import java.util.*;
public class ArrayList<E> extends AbstractList<E> { ... }  
```

An abstract class provides names and parameter types of methods that its non-abstract subclasses must implement, but not the method bodies themselves. This allows its subclasses to have completely different implementations, but be interchangeable.
/* Output reduced positive fractions, i.e., positive rationals. */
ArrayList<Rational> reduced = new ArrayList();
int d = 0;
while ( true ) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        Rational z = new Rational(r+1, c+1);
        if ( !reduced.contains(z) ) {
            System.out.println( z );
            reduced.add(z);
        }
        r--;
    }
    d++;
}
/* Output reduced positive fractions, i.e., positive rationals. */
HashSet<Rational> reduced = new HashSet();
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        Rational z = new Rational(r+1, c+1);
        if (!reduced.contains(z)) {
            System.out.println(z);
            reduced.add(z);
        }
        r--;
    }
    d++;
}
Enumeration of rationals: and provide a hash function for Pair\(<K,V>\).  

class Pair\(<K,V>\) {
   ...
   /* HashFunction. */
   @Override
   public int hashCode() {
      return key.hashCode() + value.hashCode();
   } /* hashCode */
} /* Pair */

We define a simple hash function for a pair that just sums the hash values of its constituent fields.
Enumeration of rationals: Contrast performance of ArrayList and HashSet.

/* Output reduced fractions, i.e., positive rationals; no repeats. */
public static void timing() {
    HashSet<Rational> reduced = new HashSet();
    long startTime = System.currentTimeMillis();
    int rCount = 0;  // # of rationals so far.
    int d = 0;
    while ( rCount<100000 ) {
        int r = d;
        for (int c=0; c<=d; c++) {
            /* Let z be the reduced form of the fraction (r+1)/(c+1). */
            Rational z = new Rational(r+1, c+1);
            if ( !reduced.contains(z) ) {
                /* System.out.println( z ); */
                reduced.add(z);
                rCount++;
                if ( rCount%10000==0 )
                    System.out.println( System.currentTimeMillis()-startTime );
            }
            r--;
        }
        d++;
    }
} /* timing */

Comment out the output statement so that it is not timed. Then, time every 10,000 collection insertions.
Enumeration of rationals: performance of ArrayList vs. HashSet.

Performance of ArrayList is quadratic; performance of HashSet is linear.
Enumeration of rationals: On reflection, why are we bothering to maintain the collection of already-output rationals in the first place? Why not just output \((r+1)/(c+1)\) when it is a reduced fraction?

The test for \(n/d\) being reduced is just \(gcd(n,d)==1\)?

Analyze first.
/* Output reduced fractions, i.e., positive rationals; no repeats. */
public static void timing() {
    long startTime = System.currentTimeMillis();
    int rCount = 0;  // # of rationals so far.
    int d = 0;
    while ( rCount<100000 ) {
        int r = d;
        for (int c=0; c<=d; c++) {
            if ( Rational.gcd(r+1,c+1)==1 ) {
                /* Let z be the reduced form of the fraction (r+1)/(c+1). */
                Rational z = new Rational(r+1, c+1);
                /* System.out.println( z ); */
                rCount++;
                if ( rCount%10000==0 )
                    System.out.println( System.currentTimeMillis()-startTime );
            }
            r--;
        }
        d++;
    }
} /* timing */
Enumeration of rationals: Contrast performance of ArrayList, HashSet, and gcd==1.

Performance of ArrayList is quadratic, while performance of HashSet is linear. But in contrast, checking whether the fraction is in reduced form is practically instantaneous.
Enumerating a Collection: A small lose end.

Recall that one of the operations of a collection is to enumerate its elements. This is easy when we have direct access to the collection’s implementation, e.g.,

```c
/* Enumerate items of a collection implemented as a list \(\langle A, size, n \rangle\). */
for (int k=0; k<size; k++) /* Do whatever for A[k]. */
```

```c
/* Enumerate items of a collection implemented as a histogram \(H[0..maxValue]\). */
for (int k=0; k<=maxValue; k++)
    for (int j=1; j<=H[k]; j++)
        /* Do whatever for k. */
```

But how can you enumerate the items of a collection when its implementation is hidden within a class? Specifically, how can your code be independent of the collection’s implementation?
Enumerating a Collection: A small lose end.

Let $C<E>$ be a generic subclass of $\text{AbstractCollection}<E>$. Let $c$ be an object of an instantiation of $C<E>$ for some specific element type $\text{EL}$. Then $c$ is a collection of $\text{EL}$ items, where the collection implementation is defined by $C<E>$.

An iterator for $c$ is an object $i$ that provides two methods:

- $i$.hasNext(), which returns a boolean that says whether the $i$ can be pumped for yet another element of $c$.
- $i$.next(), which returns a value of type $\text{EL}$. Provided $i$.hasNext() has just returned true, invoking $i$.next() returns the “next” element of collection $c$, where the order of enumeration is beyond your control.

N.B. Although not technically accurate, you can think of there being a generic class $\text{Iterator}<E>$ that has an instantiation $\text{Iterator}<\text{EL}>$, and $i$ is an object of that class.
Enumerating a Collection: A small lose end.

The following code pattern can be used to pump collection c for elements until there are no more:

```java
Iterator<EL> i = c.iterator();
while ( i.hasNext() ) {
    EL e = i.next();
    /* process element e. */
}
```
Enumerating a Collection: A small lose end.

Suppose after having enumerated 100,000 rationals and storing them in `reduced`, you wanted to read them out from `reduced` and process them. Then you could do so with this instance of the code pattern above:

```java
Iterator<Rational> i = reduced.iterator();
while ( i.hasNext() ) {
    Rational e = i.next();
    /* process element e. */
}
```

This code would work regardless of whether `reduced` is implemented as an `ArrayList<Rational>` or a `HashSet<Rational>`. The details of how items are extracted from `reduced` are hidden in the implementation of the particular iterator `i` that is returned by `reduced.iterator()`.