Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

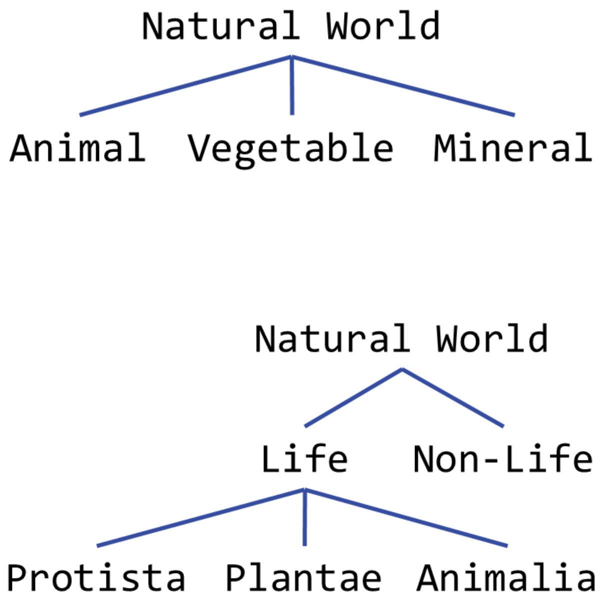
Classes and Objects

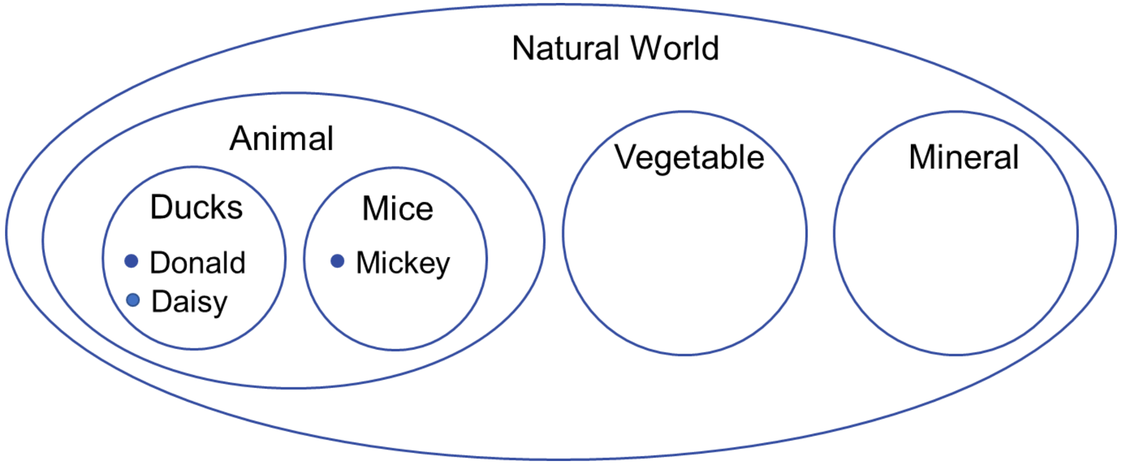
Introduction

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.





Introduction

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 in the first place, and obtain a still-faster implementation without them.

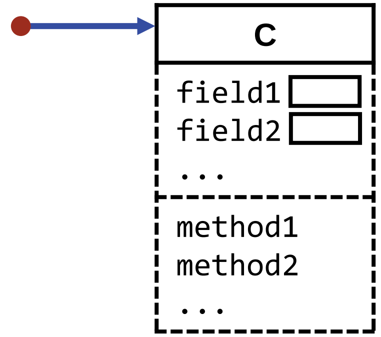
```
 Object
   |
  Pair
   |
Fraction
   |
Rational
```

Essential Notions

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.

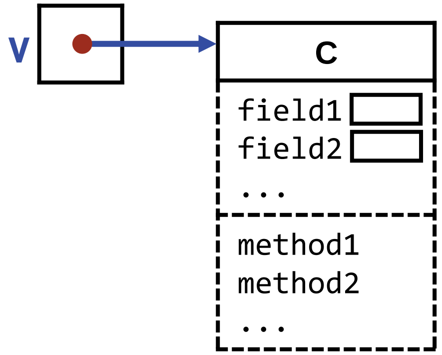


Classes are *types*. If C is a class, a variable v of type C is obtained by executing the declaration:

```
C v = expression;
```

That is, variable v (with type C) is initialized with the value of the *expression*.

Such a variable can hold a reference to an object of type C.



Essential Notions

An object *o* of type `C` is created by executing the expression

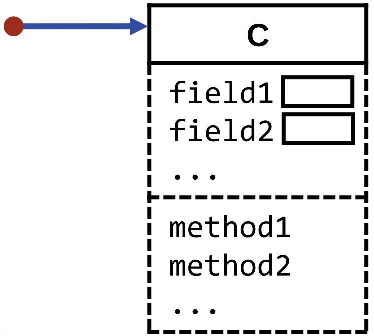
`new` `C(…)`

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

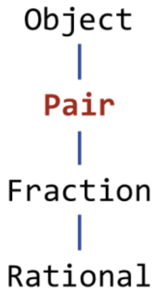
If object *o* has a method `m`, the method is invoked by *`o`*`.m(…)`.

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).



Pair

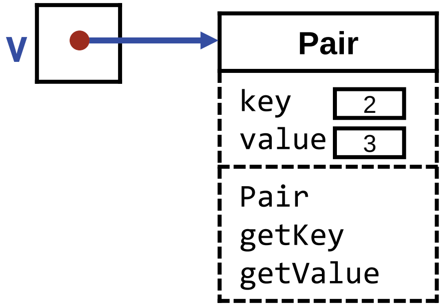

Object
Pair
Fraction
Rational

Class definition: Pair

```
class Pair {
   /* Representation. */
      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);
```



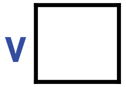
Object
|
Pair
|
Fraction
|
Rational

Execution of the variable declaration (with initialization) in four steps:

1. Create the variable v.

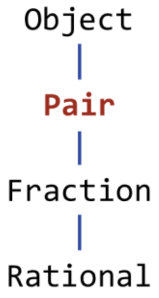
Class definition:

```
class Pair {
   /* Representation. */
      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);
```



Execution of the variable declaration (with initialization) in four steps:

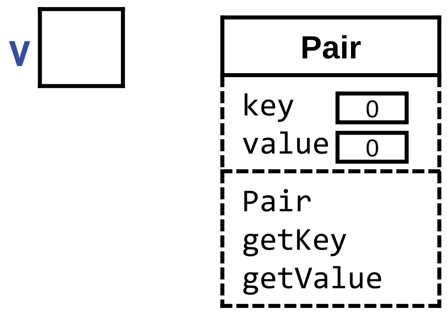
1. Create the variable v.
2. Create an object of type `Pair`.

Class definition:

```
class Pair {
   /* Representation. */
      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);
```



Pair

`Object`
`|`
`Pair`
`|`
`Fraction`
`|`
`Rational`

Execution of the variable declaration (with initialization) in four steps:

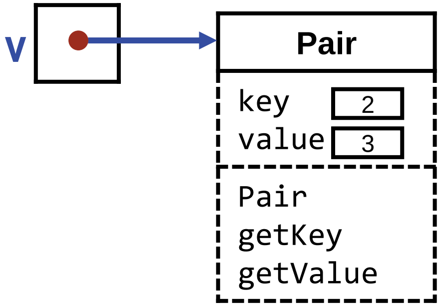
1. Create the variable v.
2. Create an object of type `Pair`.
3. Invoke the constructor `Pair` on the object, which re-initialize fields.

Class definition:

```
class Pair {
   /* Representation. */
      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);
```



Pair

Object

Pair

Fraction

Rational

Execution of the variable declaration (with initialization) in four steps:

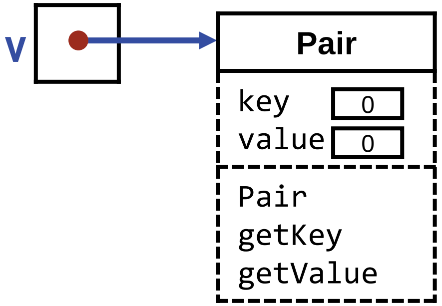
1. Create the variable v.
2. Create an object of type `Pair`.
3. Invoke the constructor `Pair` on the object, which re-initialize fields.
4. Assign a reference to the object in v.

Class definition:

```
class Pair {
   /* Representation. */
      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);
```



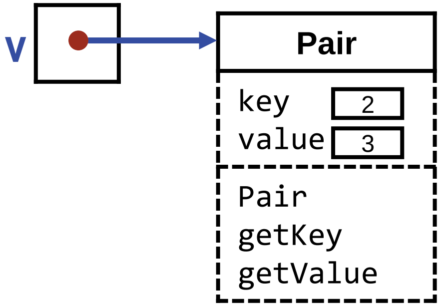
Pair

Object
|
Pair
|
Fraction
|
Rational

Visibility: Each field and method of a class has visibility `public`, `private`, or `protected`.

```
class Pair {
   /* Representation. */
      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`.



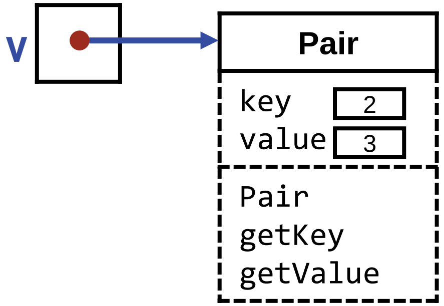
Pair

Object
Pair
Fraction
Rational

Modifiability: A **private** or **protected** field with a **public** getter is *read-only* outside its scope.

```
class Pair {
   /* Representation. */
      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*.



Pair

`Object`
`|`
`Pair`
`|`
`Fraction`
`|`
`Rational`

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:

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

```
Pair@20293791
```

Pair

Object
|
Pair
|
Fraction
|
Rational

Overriding definition of `toString` for `Pair`:

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

Output the `String` representation of an object:

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

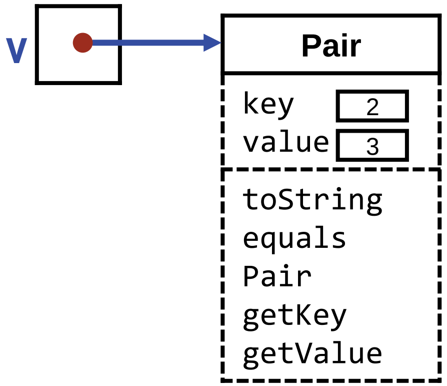
```
<2,3>
```

Object
|
Pair
|
Fraction
|
Rational

Execution of the print statement in three steps:

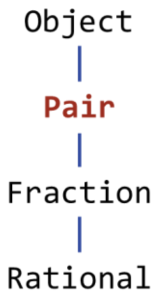
Overriding definition of `toString` for `Pair`:

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



Output the `String` representation of an object:

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



Execution of the print statement in three steps:

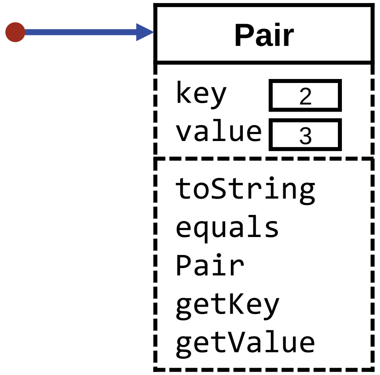
1. Obtain the value of variable v.

Overriding definition of `toString` for `Pair`:

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

Output the `String` representation of an object:

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



Pair

Object
Pair
Fraction
Rational

Execution of the print statement in three steps:

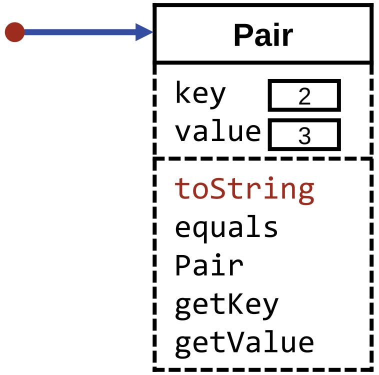
1. Obtain the value of variable v.
2. Compute the `String` representation of that value by invoking its `toString` method.

Overriding definition of `toString` for `Pair`:

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

Output the `String` representation of an object:

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



Object
|
Pair
|
Fraction
|
Rational

Execution of the print statement in three steps:

1. Obtain the value of variable v.
2. Compute the `String` representation of that value by invoking its `toString` method.
3. Output that value.

Overriding definition of `toString` for `Pair`:

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

Output the `String` representation of an object:

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

```
<2,3>
```

Pair

Object
Pair
Fraction
Rational

Execution of the print statement in three steps:

1. Obtain the value of variable v.
2. Compute the `String` representation of that value by invoking its `toString` method.
3. Output that value.

Overriding definition of `toString` for `Pair`:

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

A subtlety noted in passing:
What exactly is going on when **`int`** values (like `key` and `value`) are concatenated with `String` values (like `"<"`, `","`, and `">"`)? We will finesses this question. Note, however, that it also involves obtaining a `String` representation (this time, a decimal numeral) from another type of value (a 32-bit fixed-point **`int`**).

Output the `String` representation of an object:

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

```
<2,3>
```

Pair

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);
```

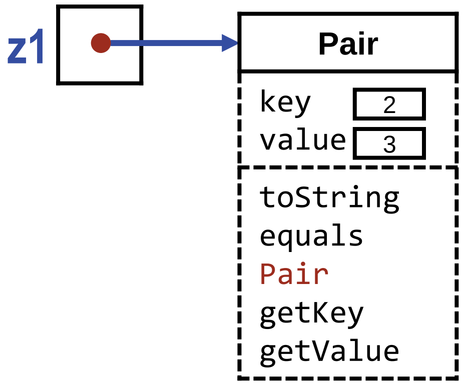
```
false
true
```

Pair

Object
|
Pair
|
Fraction
|
Rational

The definition of operator == for objects is identity.

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



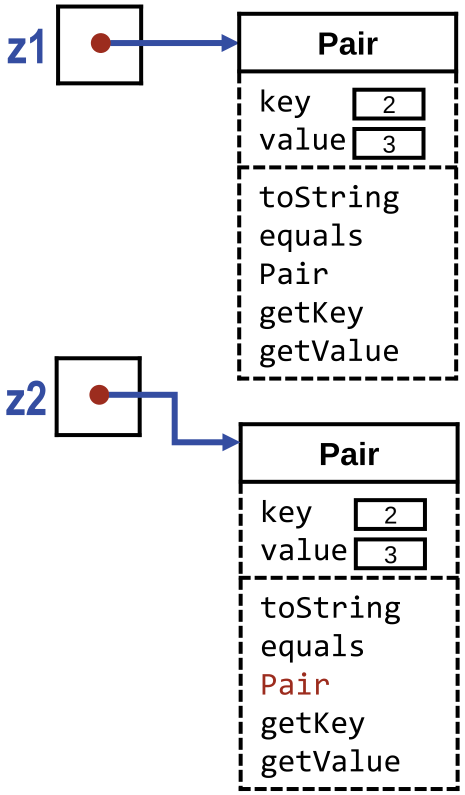
Demonstrate the difference between identity and equality.

```
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);
```

`Object`
`|`
`Pair`
`|`
`Fraction`
`|`
`Rational`

The definition of operator == for objects is identity.

- "Identity" means "exactly the same object".



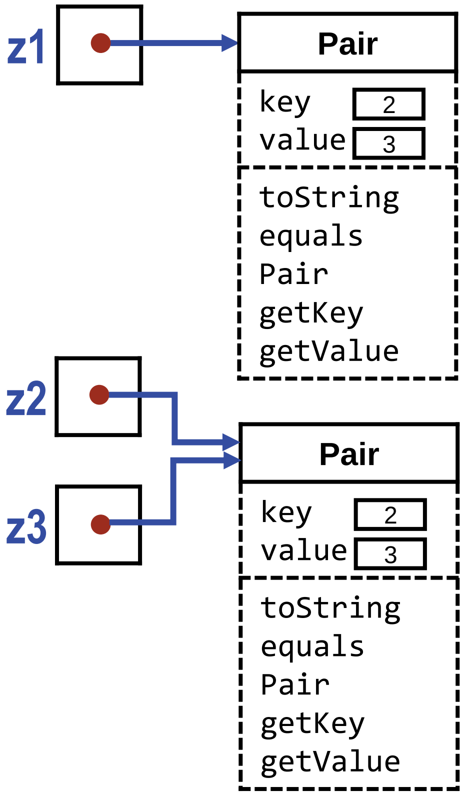
Demonstrate the difference between identity and equality.

```
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);
```

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.

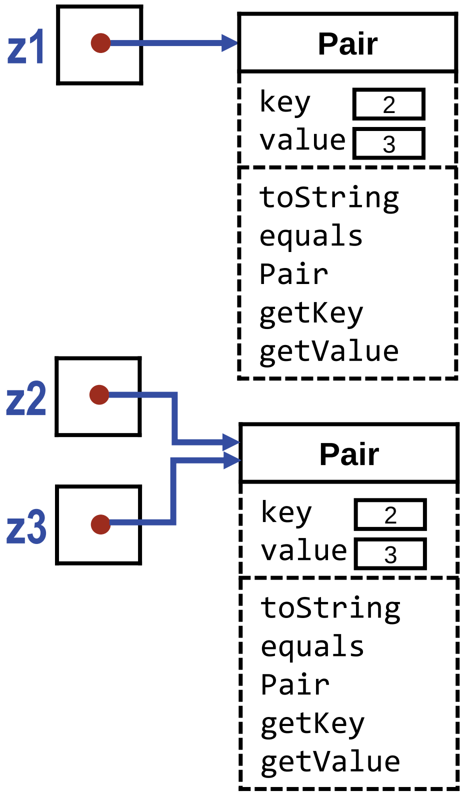
```
Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);
System.out.println(z2==z3);
```

Pair

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.

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

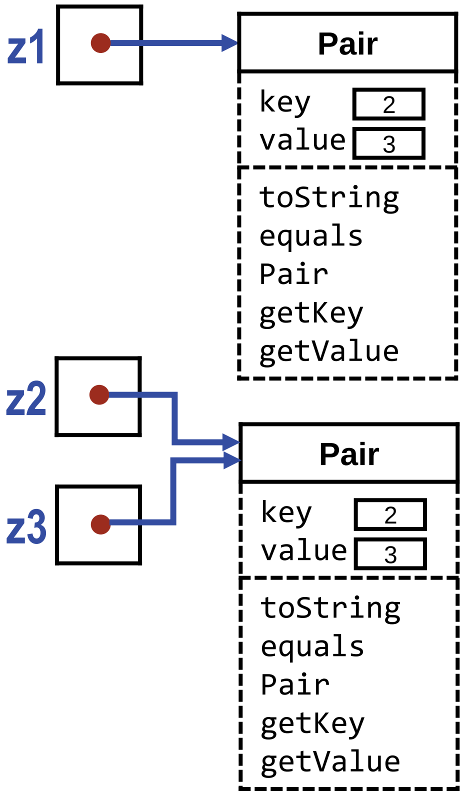
```
false
```

Pair

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.

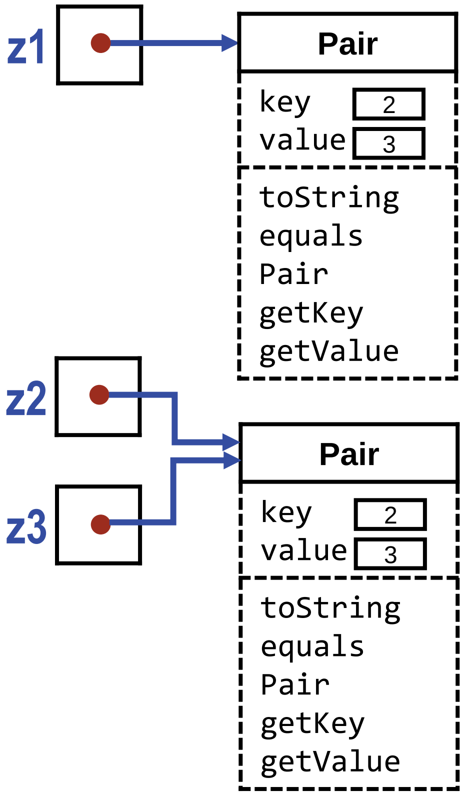
```
Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1==z2);
System.out.println(z2==z3);
```

```
false
true
```

Object
|
Pair
|
Fraction
|
Rational

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 ==.



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));
System.out.println(z2.equals(z3));
```

with the default definition of equals, i.e., identity

```
false
true
```

`Object`
`Pair`
`Fraction`
`Rational`

The default definition of `equals` can be overridden.

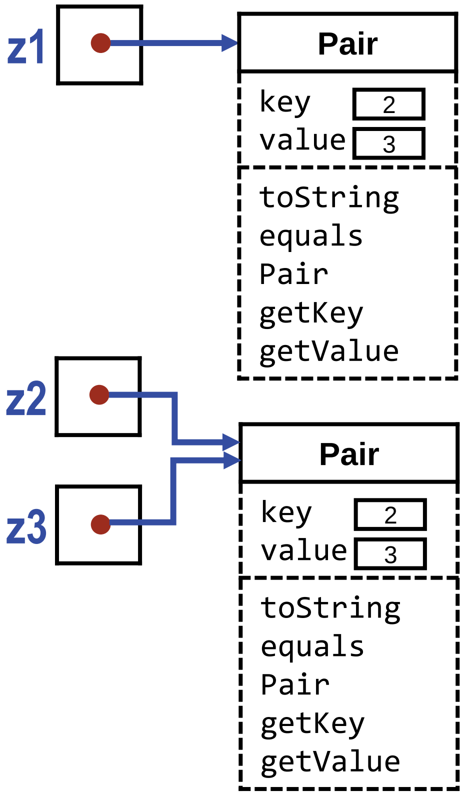
- “Identity” means “exactly the same object”.
- Every `Object` has an `equals` method that can be applied to another `Object` to test “equality”, which is user-definable.
- The default definition of method `equals` in `Object` is identity, i.e., the same as ==.
- Unlike the == operator, `equals` can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

Demonstrate the difference between identity and equality.

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

with the overriding definition
of `equals` shown on the next slide

```
true
true
```



Pair

Object
Pair
Fraction
Rational

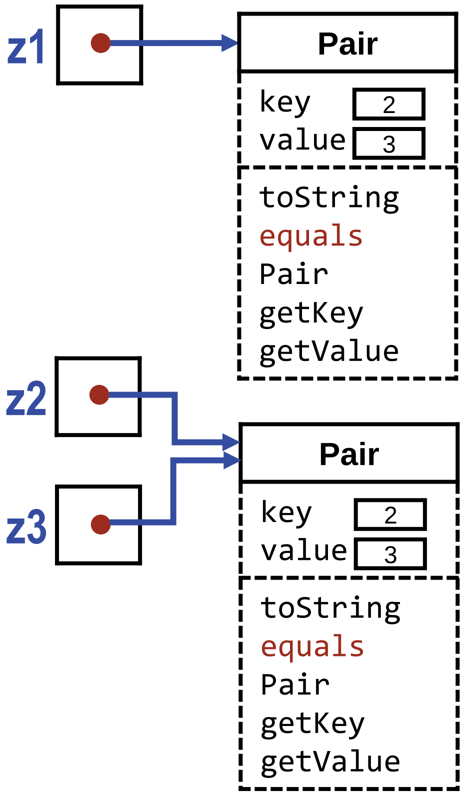
Demonstrate the difference between identity and equality.

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

```
true
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 */
```



Asks the compiler to warn if the next method definition is not overriding.

Pair

Object
|
Pair
|
Fraction
|
Rational

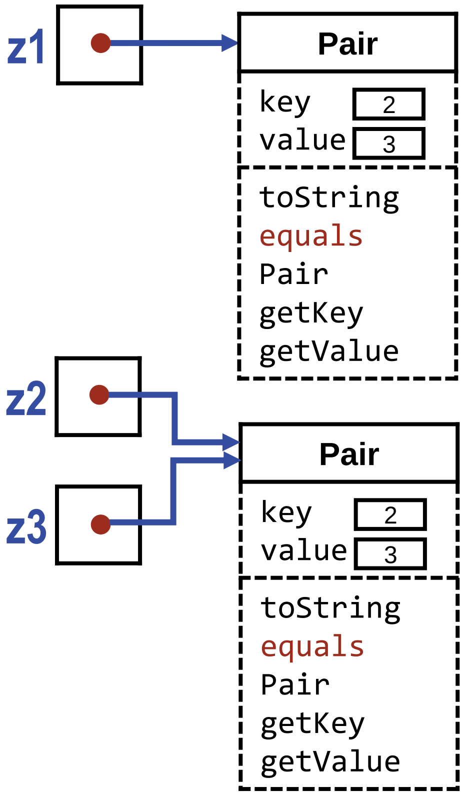
Demonstrate the difference between identity and equality.

```
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 */
```

An Object is never equal to no Object.



Object
|
Pair
|
Fraction
|
Rational

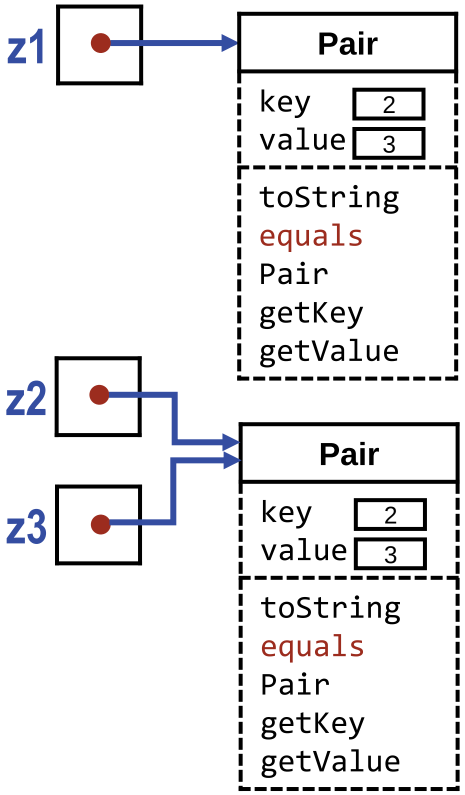
Demonstrate the difference between identity and equality.

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

```
true
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 */
```



An `Object` is always equal to itself, e.g. `z2` and `z3`.

Pair

```
Object
  |
 Pair
  |
Fraction
  |
Rational
```

Demonstrate the difference between identity and equality.

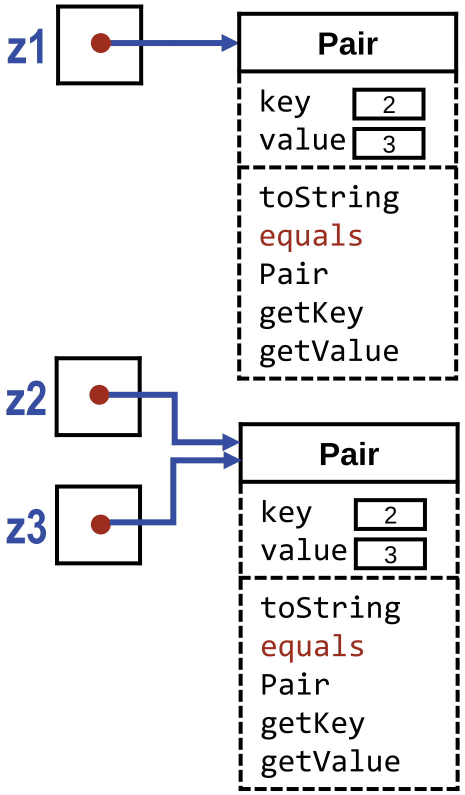
```
Pair z1 = new Pair(2,3);
Pair z2 = new Pair(2,3);
Pair z3 = z2;
System.out.println(z1.equals(z2));
System.out.println(z2.equals(z3));
```

```
true
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.



```
Object
  |
 Pair
  |
Fraction
  |
Rational
```

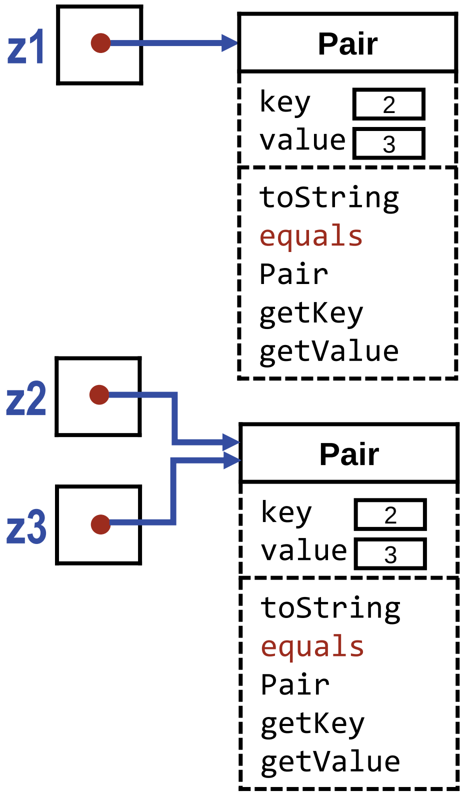
Demonstrate the difference between identity and equality.

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

```
true
true
```

Overriding definition of equals for pairs.

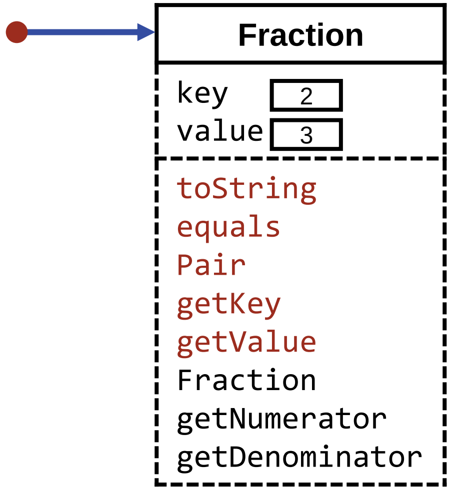
```
class Pair {
   ...
   /* Equality. */
      @Override
      public boolean equals(Object q) {
         if (q==null) return false;
         if (q==this) return true;
         if ( !(q instanceof Pair) ) return false;
         Pair qPair = (Pair)q;
         return (key == qPair.key) && (value == qPair.value);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



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

Object
|
Pair
|
Fraction
|
Rational

Fraction is a subclass of Pair, and as such acquires the fields and methods from Pair.


Fraction

key 2
value 3

toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

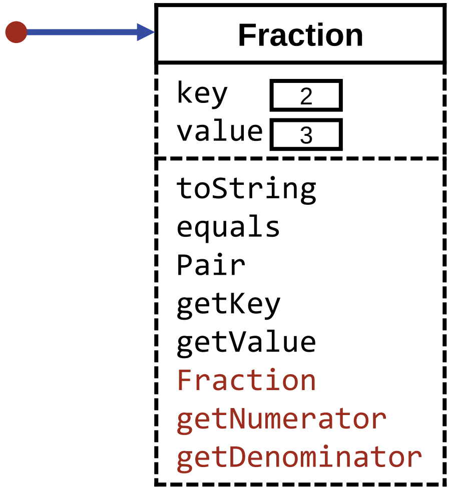
Fraction

Subclass definition: 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 */
```

Object
|
Pair
|
Fraction
|
Rational

Fraction is a subclass of Pair, and as such acquires the fields and methods from Pair, while adding more of its own.


Fraction

key 2
value 3

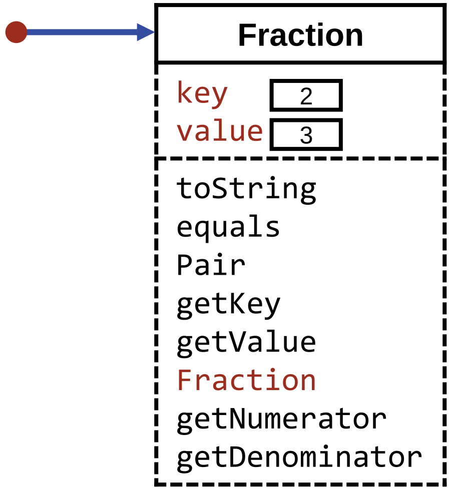
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

Subclass definition: 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 */
```

Fraction

Object
|
Pair
|
Fraction
|
Rational



Fraction

key 2
value 3

toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

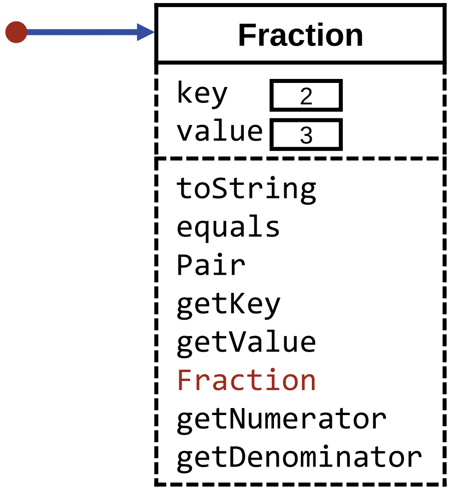
Subclass definition: `Fraction`

```
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 */
```

The `Fraction` constructor uses the `Pair` constructor to set fields key and value.

Fraction

Object
|
Pair
|
Fraction
|
Rational


Fraction
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

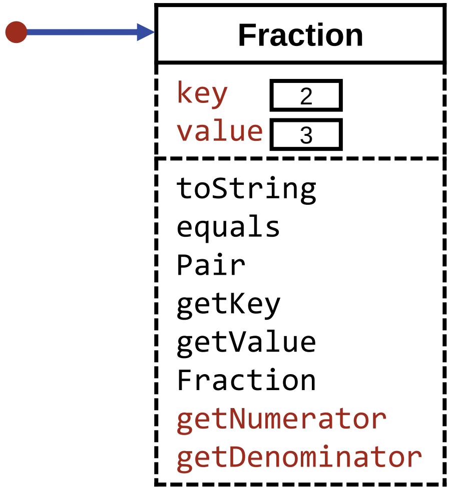
Subclass definition: `Fraction`

```
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 */s
```

It then assures that the `denominator` is not zero.

Fraction

Object
Pair
Fraction
Rational

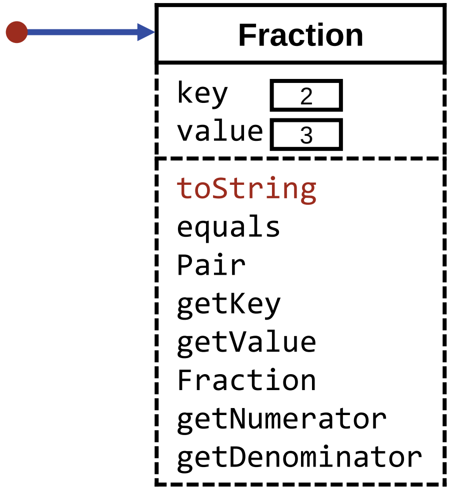

Fraction
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

Subclass definition: `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 */
```

Getters have direct access to the fields `key` and `value` because they are declared **`protected`** in `Pair`, a superclass of `Fraction`.

Object
Pair
Fraction
Rational



Overriding definition of `toString` for `Fraction`:

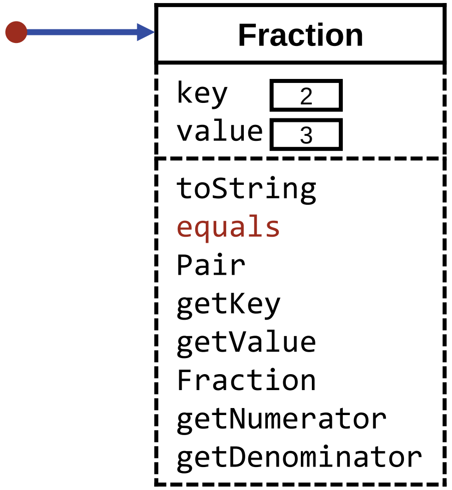
```
class Fraction extends Pair {
   /* Constructor. */
      public Fraction(int numerator, int denominator) {
         super(numerator, denominator); // Apply the Pair constructor.
         assert denominator!=0: "0 denominator";
         }

   /* Access. */
      public int getNumerator()   { return key; }
      public int getDenominator() { return value; }

   /* String representation. */
      public String toString() { return key + "/" + value; }
   } /* Fraction */
```

Different string representations for `Pair` and `Fraction`

```
System.out.println(Pair(2,3) + " " + Fraction(2,3));
```

```
<2,3> 2/3
```

Object
|
Pair
|
Fraction
|
Rational



Overriding definition of `equals` for `Fraction`:

Two fractions are equal iff they have equal numerators and equal denominators. This is (almost) the test that is used to test the equality of two `Pairs`, so we might consider omitting an overriding definition of `equals` for `Fraction`, and rely on the definition in `Pair`.

However, pairs and fractions are two fundamentally different sorts of things, and it seems inappropriate to let a `Fraction` be considered equal to a `Pair` just because they happen to have the same two equal fields. A `Fraction` uses a `Pair` for its representation more as a convenience than because fractions are a special sort of pair.

Were `Fraction` to rely on the definition of `equals` in `Pair`

```
Pair z1 = Pair(2,3);
Fraction z2 = Fraction(2,3);
System.out.println(z1 + " " + z2);
System.out.println(z1==z2);
```

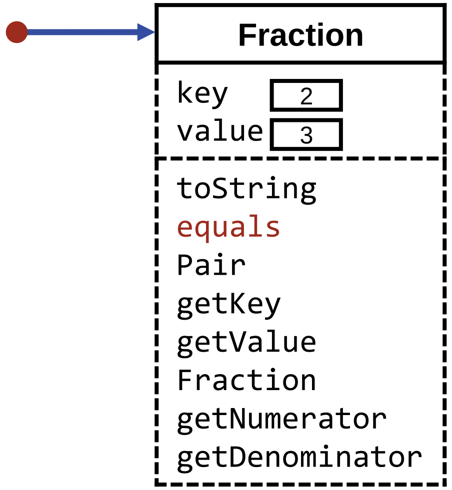
```
<2,3> 2/3
true
```

Object
Pair
Fraction
Rational

Overriding definition of equals for `Fraction`:

Accordingly, we choose to give `Fraction` its own definition of `equals`, and so treat fractions as fundamentally different from pairs.

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


Fraction
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

The effect of `Fraction` getting its own definition of equals

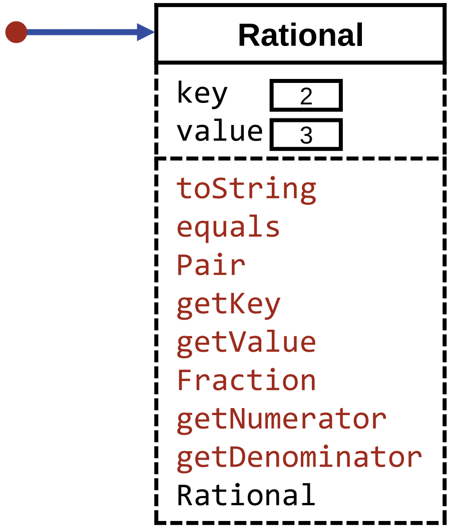
```
Pair z1 = Pair(2,3);
Fraction z2 = Fraction(2,3);
System.out.println(z1 + " " + z2);
System.out.println(z1==z2);
```

```
<2,3> 2/3
false
```

Rational

Object
|
Pair
|
Fraction
|
Rational

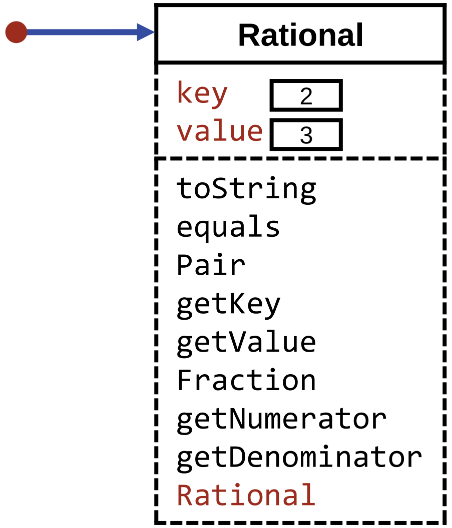
Rational is a subclass of Fraction, and as such acquires fields and methods of a Fraction.



Subclass definition: Rational

```
class Rational extends Fraction {
   ________
   } /* Rational */
```

Object
|
Pair
|
Fraction
|
Rational



Subclass definition: `Rational`

```
class Rational extends Fraction {
   /* Constructor. */
      public Rational(int numerator, int denominator) {
         super(numerator, denominator);    // Apply the Fraction constructor.
         ________
         }
   } /* Rational */
```

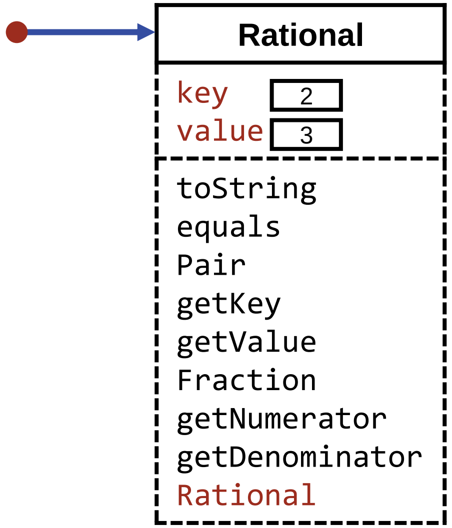
The `Rational` constructor uses the `Fraction` constructor to set fields key and value, and to check that the denominator is not zero.

Rational

Object
Pair
Fraction
Rational

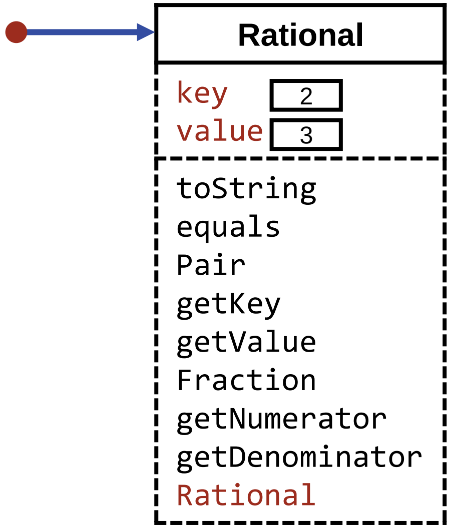
Subclass definition: Rational

```
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 */
```


Rational
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

The Rational constructor uses the Fraction constructor to set fields key and value, and to check that the denominator is not zero. Then it updates the representation to reduced form, i.e., no common factors.

Object
|
Pair
|
Fraction
|
Rational



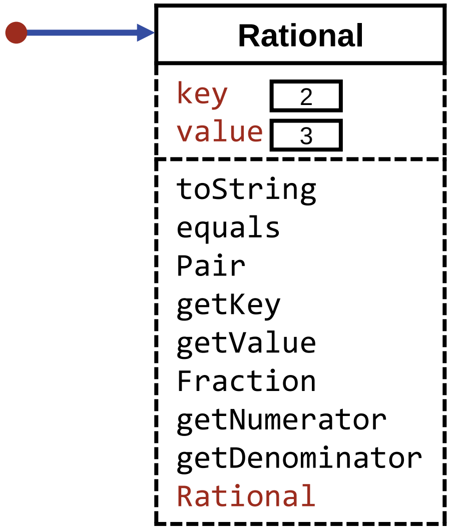
Subclass definition: `Rational`

```
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 */
```

☞ **Boundary conditions. Dead last, but don't forget them.**

Rational

Object
|
Pair
|
Fraction
|
Rational



Subclass definition: `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 */
```

Function gcd will fail for negative arguments!

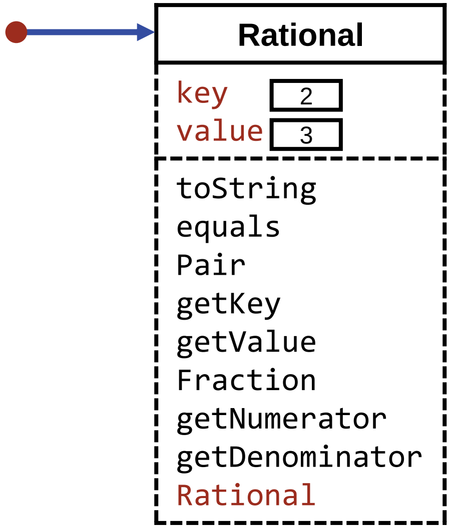
☞ **Boundary conditions. Dead last, but don't forget them.**

Object
Pair
Fraction
Rational

Subclass definition: `Rational`

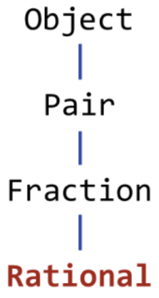
```
class Rational extends Fraction {
   /* Constructor. */
      public Rational(int numerator, int denominator) {
         super(numerator, denominator);    // Apply the Fraction constructor.
         int g = gcd(numerator, denominator);
         key = numerator/g;
         value = denominator/g;
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

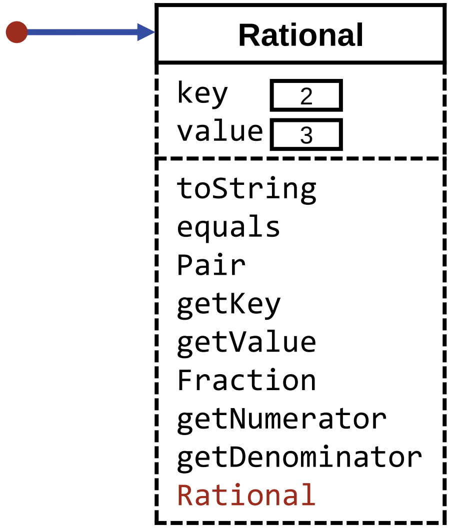
Function gcd will fail for non-positive arguments!



Rational
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

☞ **Boundary conditions. Dead last, but don't forget them.**





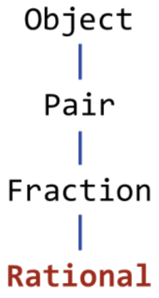
Subclass definition: `Rational`

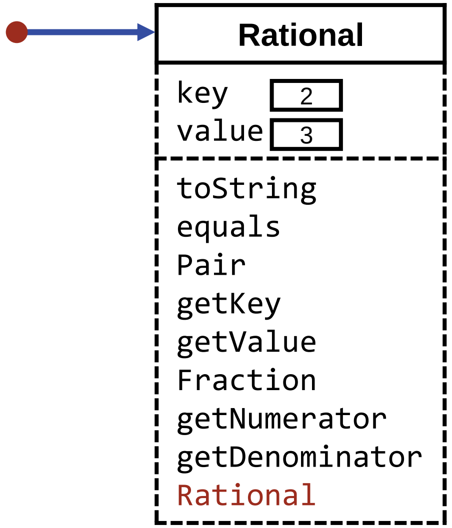
```
class Rational extends Fraction {
   /* Constructor. */
      public Rational(int numerator, int denominator) {
         super(numerator, denominator);    // Apply the Fraction constructor.
         int g = gcd(numerator, denominator);
         key = numerator/g;
         value = denominator/g;
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

Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

☞ **Boundary conditions. Dead last, but don't forget them.**

Rational


Object
Pair
Fraction
Rational


Rational
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

Subclass definition: `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 */
```

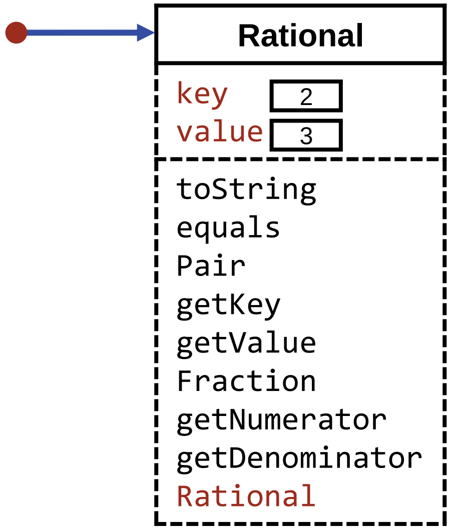
Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

- Zero should have a denominator of 1.
- Negatives should have a negative numerator and a positive denominator.
- Positives should have positive numerator and denominator.

☞ **Boundary conditions. Dead last, but don't forget them.**

Rational

Object
Pair
Fraction
Rational



Rational

key 2
value 3

toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

Subclass definition: `Rational`

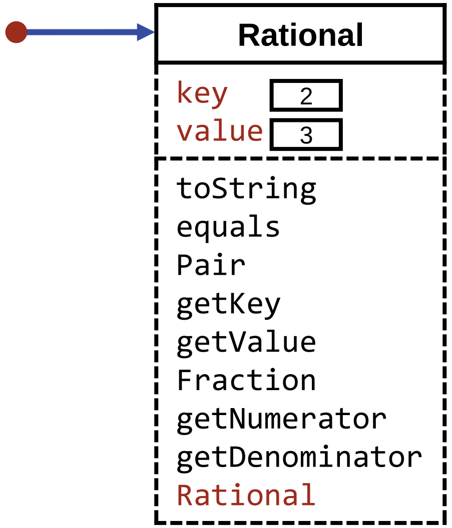
```
class Rational extends Fraction {
   /* Constructor. */
      public Rational(int numerator, int denominator) {
         super(numerator, denominator);    // Apply the Fraction constructor.
         if ( numerator==0 ) value = 1;
         __________
         }
   } /* Rational */
```

Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

- Zero should have a denominator of 1.

☞ **Boundary conditions. Dead last, but don't forget them.**

Object
|
Pair
|
Fraction
|
Rational



Subclass definition: `Rational`

```
class Rational extends Fraction {
   /* Constructor. */
      public Rational(int numerator, int denominator) {
         super(numerator, denominator);    // Apply the Fraction constructor.
         if ( numerator==0 ) value = 1;
         else {
            int g = gcd(abs(numerator), abs(denominator));
            if ( (numerator<0) && (denominator>0)) || (
                 (numerator>0) && (denominator<0) ) sign = -1;
            else sign = +1;
            key   = sign*abs(numerator)/g;
            value = abs(denominator)/g;
            }
         }
   } /* Rational */
```

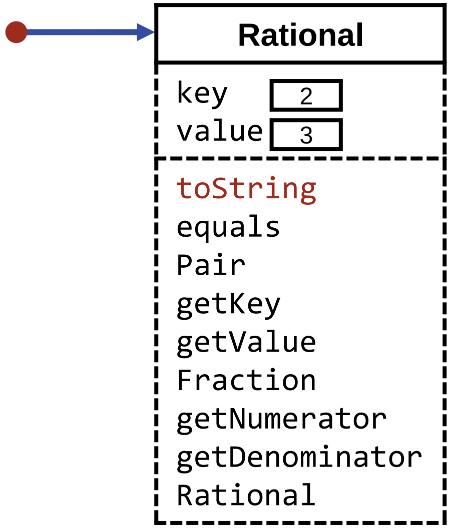
Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

- Zero should have a denominator of 1.
- Negatives should have a negative numerator and a positive denominator.
- Positives should have positive numerator and denominator.

Object
Pair
Fraction
Rational

Overriding definition of `toString` for `Rational`:

```
class Rational extends Fraction {
   ... */
   } /* Rational */

   /* String representation. */
      public String toString() {
         if ( value==1 ) return key + "";   // this as an integer
         else return super.toString();      // this as a Fraction
         } /* toString
```



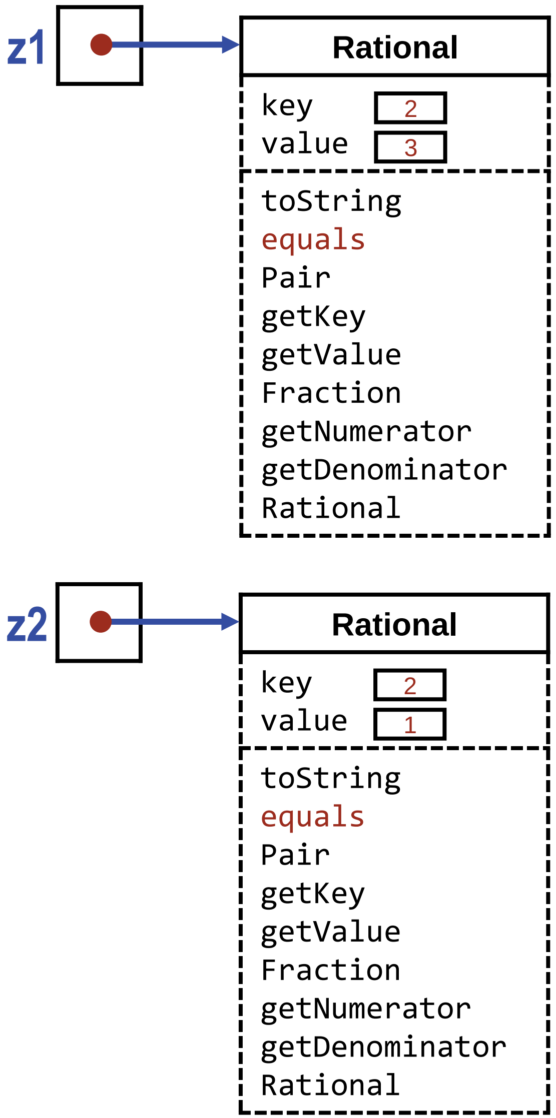
Rational

Rational

Object
|
Pair
|
Fraction
|
Rational

Overriding definition of `equals` for `Rational` is not needed:

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.



The effect of letting `Rational` rely on the definition of `equals` in `Fraction`

```
Rational z1 = Rational(4,6);
Rational z2 = Rational(6,3);
System.out.println(z1 + " " + z2);
System.out.println(z1==z2);
```

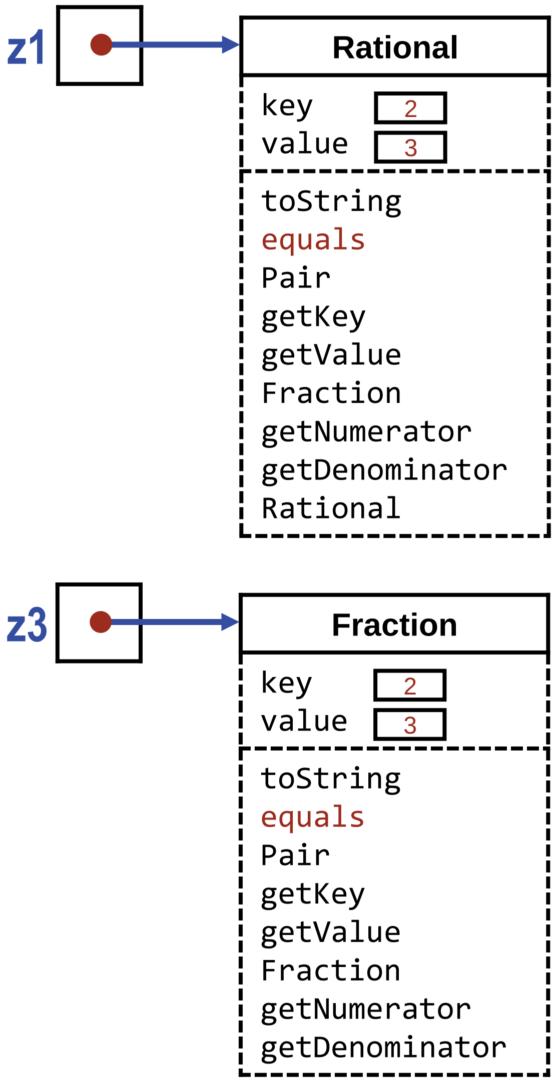
```
2/3 2
false
```

Rational

Object
|
Pair
|
Fraction
|
Rational

Overriding definition of `equals` for `Rational` is not needed:

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.



The effect of letting `Rational` rely on the definition of equals in `Fraction`

```
Rational z1 = Rational(4,6);
Fraction z3 = Fraction(2,3);
System.out.println(z1 + " " + z3);
System.out.println(z1==z3);
```

```
2/3 2/3
true
```

Rational

Object
|
Pair
|
Fraction
|
Rational

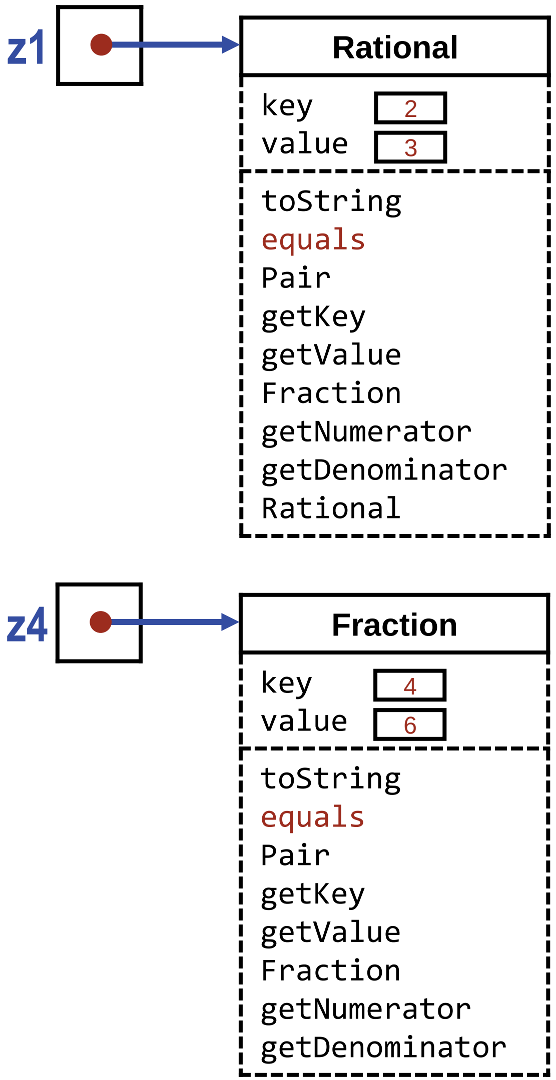
Overriding definition of `equals` for `Rational` is not needed:

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.

The effect of letting `Rational` rely on the definition of equals in `Fraction`

```
Rational z1 = Rational(4,6);
Fraction z4 = Fraction(4,6);
System.out.println(z1 + " " + z4);
System.out.println(z1==z4);
```

```
2/3 4/6
false
```



z1

Rational

key 2
value 3

toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

z4

Fraction

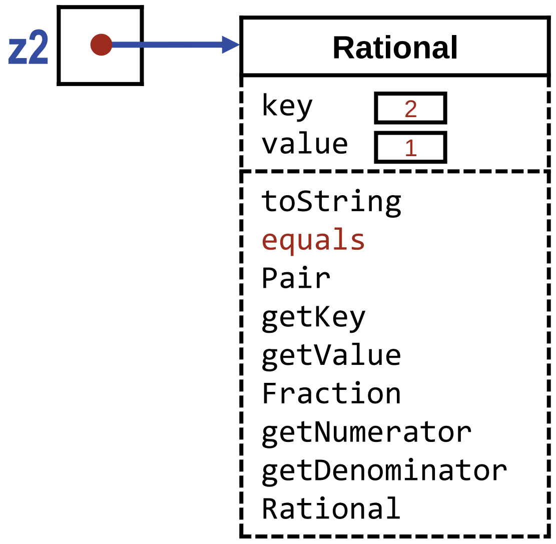
key 4
value 6

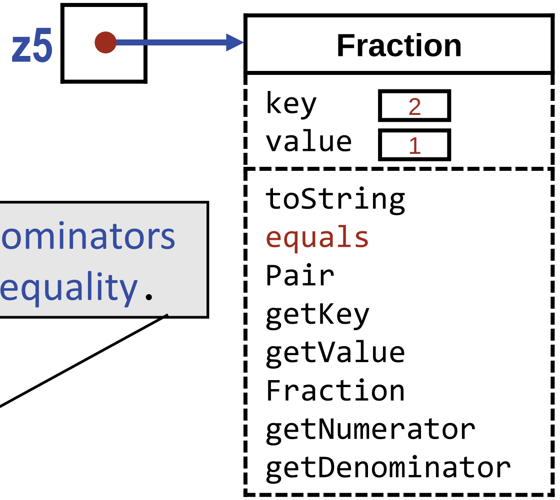
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

Object
|
Pair
|
Fraction
|
Rational

Overriding definition of `equals` for `Rational` is not needed:

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.





The display of rationals with denominators of 1 as integers has no effect on equality.

The effect of letting `Rational` rely on the definition of equals in `Fraction`

```
Rational z2 = Rational(6,3);
Fraction z5 = Fraction(2,1);
System.out.println(z2 + " " + z5);
System.out.println(z2==z5);
```

```
2 2/1
true
```

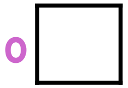
Unit Test: Cover every public aspect of the class’s interface (*black-box* testing), and if you know the implementation internals, every corner case you can foresee (*white-box* testing).

Test code	Output
`System.out.println(Rational(2,3));`	`2/3`
`System.out.println(Rational(4,6));`	`2/3`
`System.out.println(Rational(-4,6));`	`-2/3`
`System.out.println(Rational(4,-6));`	`-2/3`
`System.out.println(Rational(-4,-6));`	`2/3`
`System.out.println(Rational(6,3));`	`2`
`System.out.println(Rational(0,1));`	`0`
`System.out.println(Rational(0,10));`	`0`
`System.out.println(Rational(0,-10));`	`0`
`System.out.println(Rational(2,3)==Rational(4,6));`	`true`

Can you think of other examples to test?

Subtype Polymorphism and Dynamic Dispatch

```
Object
  |
 Pair
  |
Fraction
  |
Rational
```

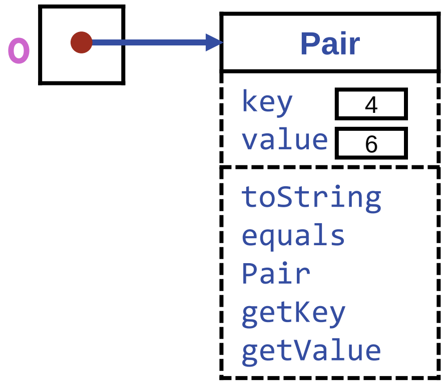


Subtype polymorphism: A variable of class C can be assigned a reference to any object of class C′, where C′ is either C itself, or C′ is a subclass of C, i.e., lower in the class hierarchy.

```
Object o;
```

Subtype Polymorphism and Dynamic Dispatch

```
Object
  |
 Pair
  |
Fraction
  |
Rational
```

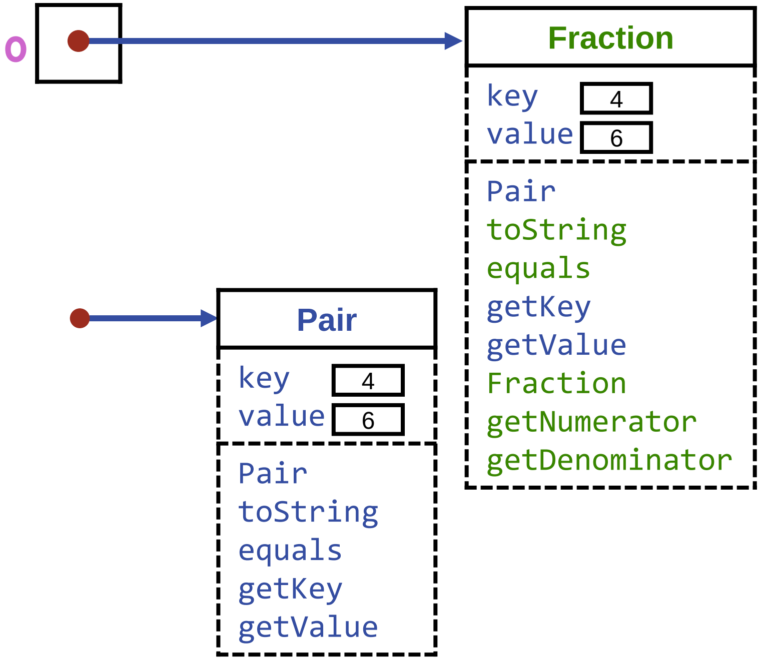


Subtype polymorphism:

```
Object o;
o = new Pair(4,6);
```

Object
|
Pair
|
Fraction
|
Rational

Subtype polymorphism:



o

Fraction
key 4
value 6
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator

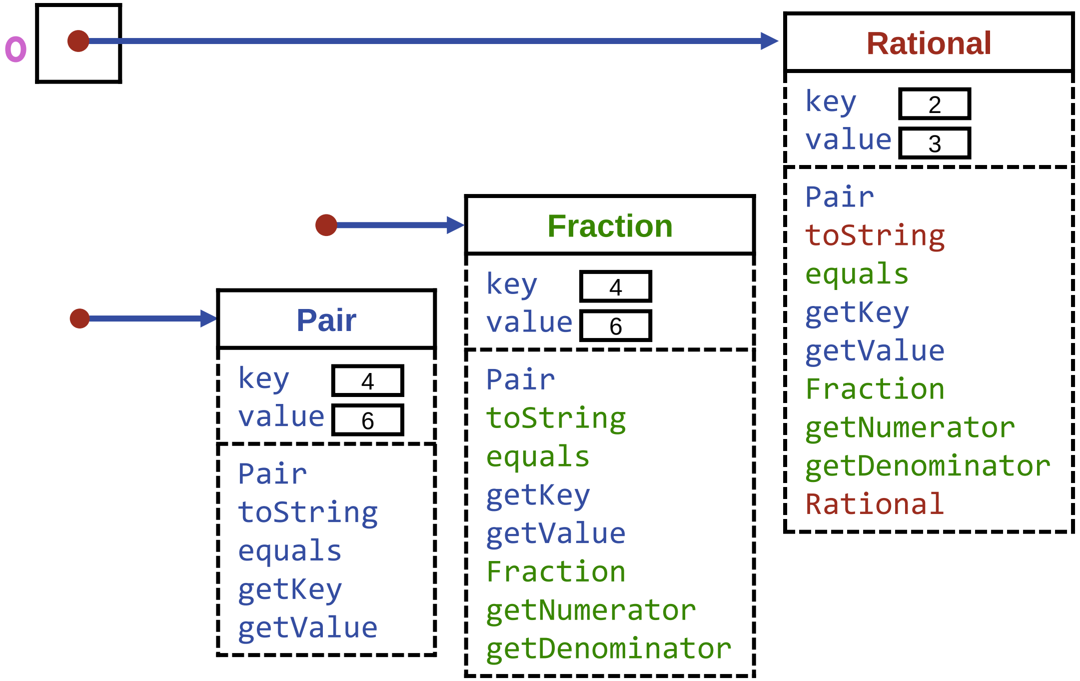
Pair
key 4
value 6
Pair
toString
equals
getKey
getValue

```
Object o;
o = new Pair(4,6);
o = new Fraction(4,6);
```

Subtype Polymorphism and Dynamic Method Dispatch

Object
|
Pair
|
Fraction
|
Rational

Subtype polymorphism:

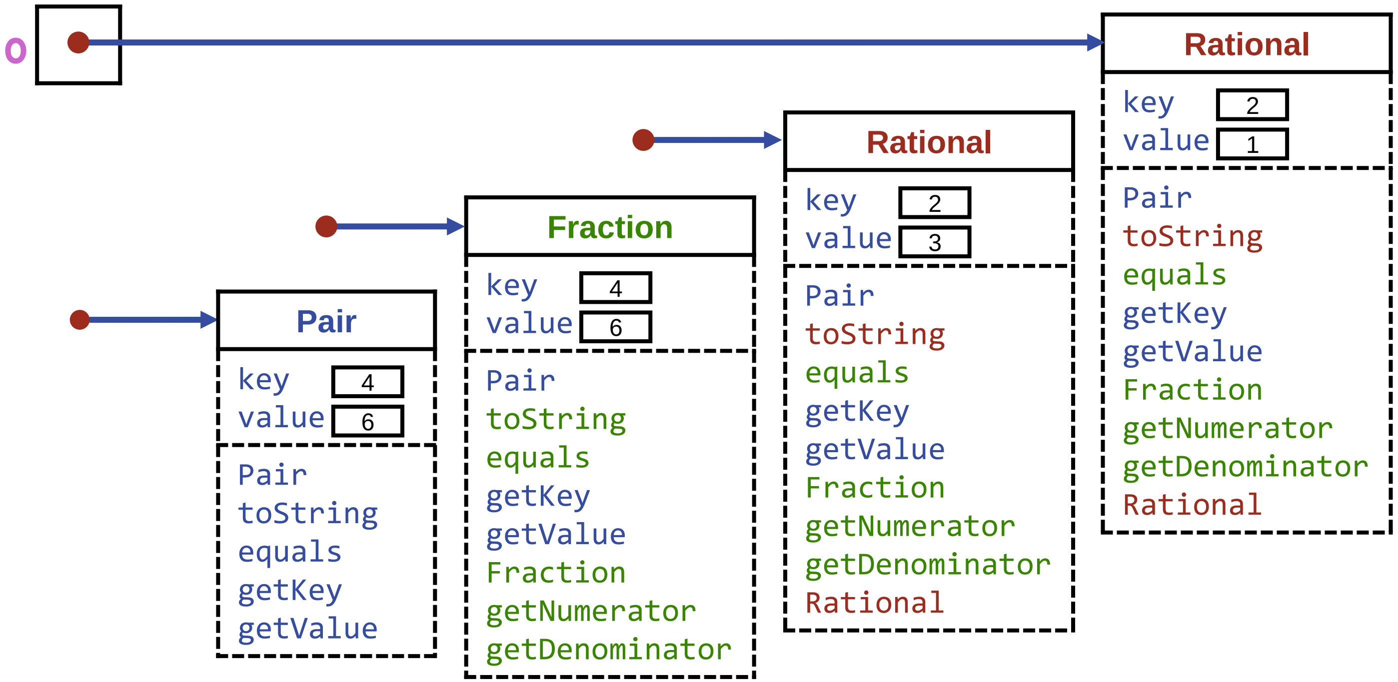


```
Object o;
o = new Pair(4,6);
o = new Fraction(4,6);
o = new Rational(4,6);
```

Subtype Polymorphism and Dynamic Method Dispatch

```
Object
  |
 Pair
  |
Fraction
  |
Rational
```

Subtype polymorphism:


o

Pair
key 4
value 6
Pair
toString
equals
getKey
getValue

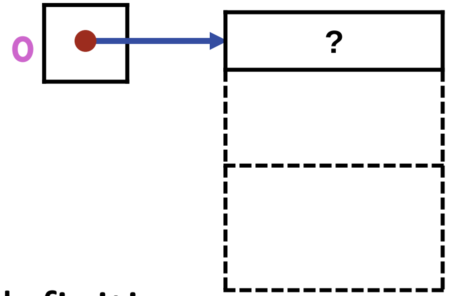
Fraction
key 4
value 6
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator

Rational
key 2
value 3
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

Rational
key 2
value 1
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

```
Object o;
o = new Pair(4,6);
o = new Fraction(4,6);
o = new Rational(4,6);
o = new Rational(6,3);
```

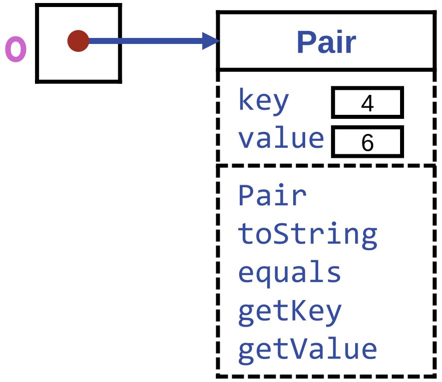
Subtype Polymorphism and Dynamic Method Dispatch



Dynamic method dispatch: The definition used for any given method invocation depends of the type of the value, not the type of the variable that contains that value.

```
Object o;
```

Object
|
Pair
|
Fraction
|
Rational



Dynamic method dispatch:

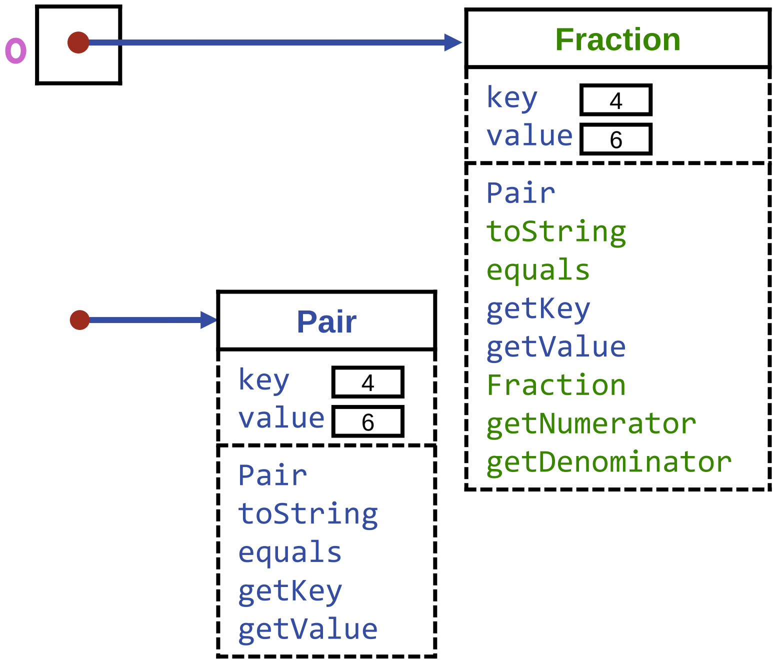
```
Object o;
o = new Pair(4,6);      System.out.println( o );
```

```
<4,6>
```

Subtype Polymorphism and Dynamic Method Dispatch

```
Object
  |
Pair
  |
Fraction
  |
Rational
```

Dynamic method dispatch:



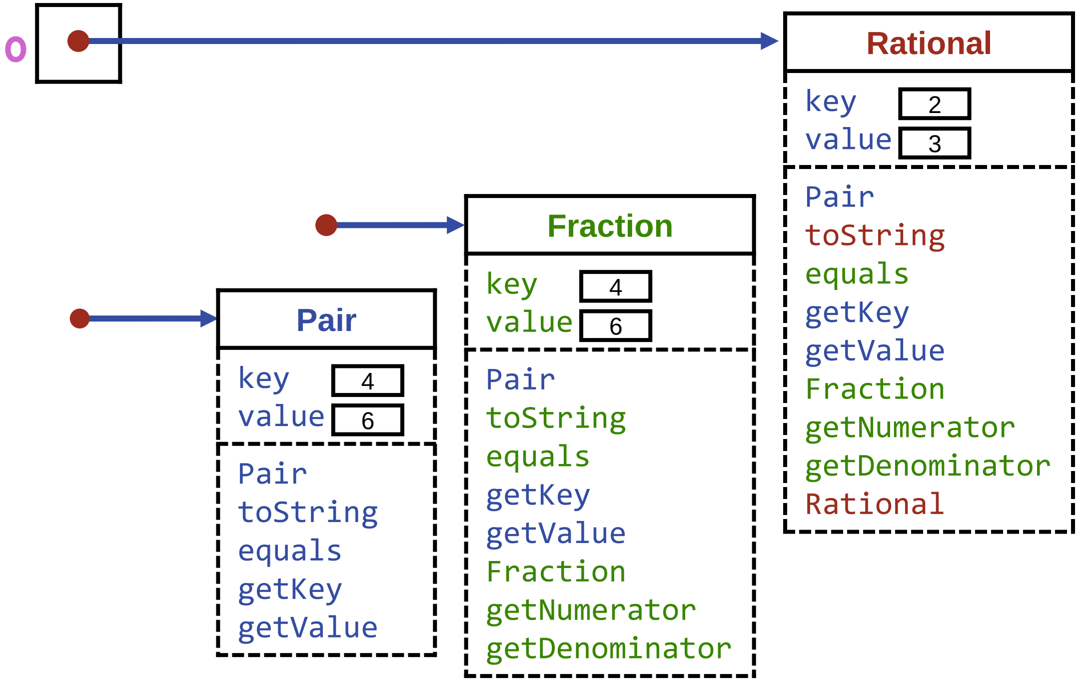
```
Object o;
o = new Pair(4,6);     System.out.println( o );
o = new Fraction(4,6); System.out.println( o );
```

```
<4,6>
4/6
```

Subtype Polymorphism and Dynamic Method Dispatch

Object
Pair
Fraction
Rational

Dynamic method dispatch:


o

Rational
key 2
value 3
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

Fraction
key 4
value 6
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator

Pair
key 4
value 6
Pair
toString
equals
getKey
getValue

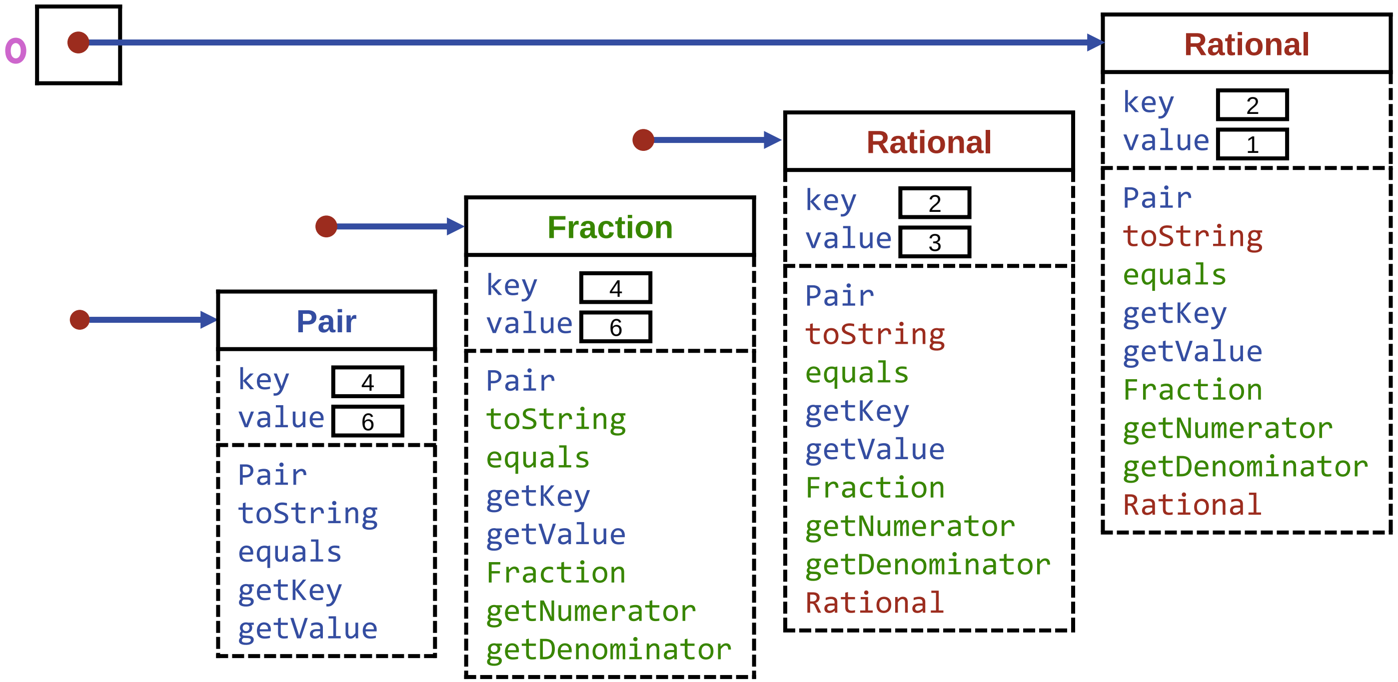
```
Object o;
o = new Pair(4,6);     System.out.println( o );
o = new Fraction(4,6); System.out.println( o );
o = new Rational(4,6); System.out.println( o );
```

```
<4,6>
4/6
2/3
```

Subtype Polymorphism and Dynamic Method Dispatch

Object
Pair
Fraction
Rational

Dynamic method dispatch:


o

Pair
key 4
value 6
Pair
toString
equals
getKey
getValue

Fraction
key 4
value 6
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator

Rational
key 2
value 3
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

Rational
key 2
value 1
Pair
toString
equals
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

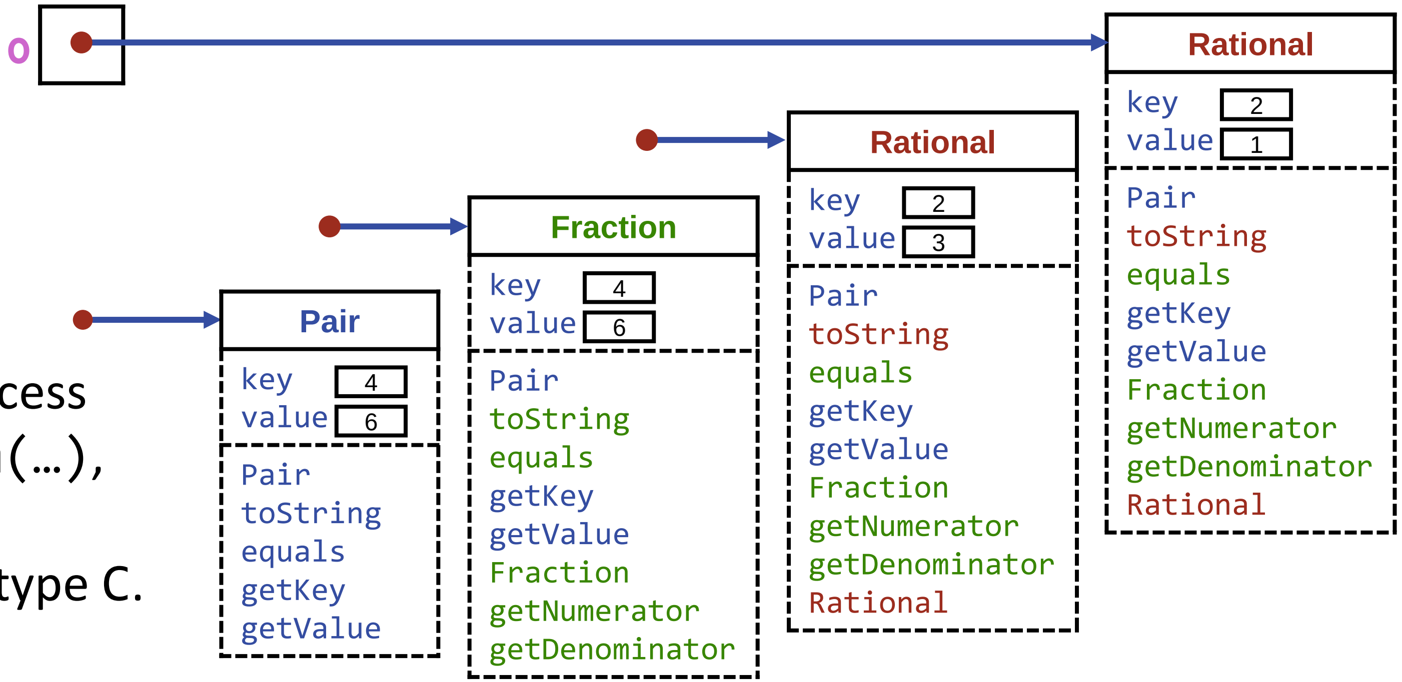
```
Object o;
o = new Pair(4,6);     System.out.println( o );
o = new Fraction(4,6); System.out.println( o );
o = new Rational(4,6); System.out.println( o );
o = new Rational(6,3); System.out.println( o );
```

```
<4,6>
4/6
2/3
2
```

Object
|
Pair
|
Fraction
|
Rational

Subtype polymorphism caveat:

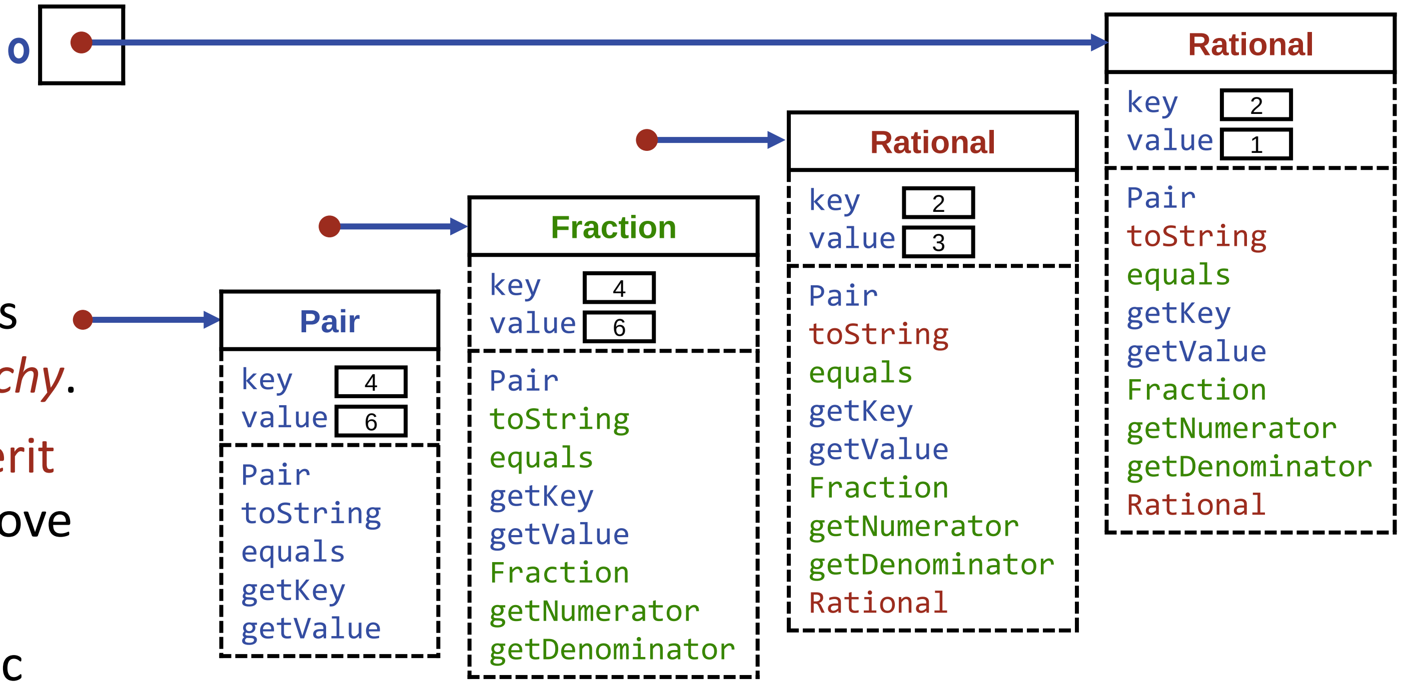
If variable v has type C, a field access `v.f`, or a method invocation `v.m(…)`, requires that field `f` or method `m` necessarily exist in any object of type C.



```
Object   o = new Pair(4,6);     System.out.println(o.getKey());        // Illegal.
Pair     p = new Pair(4,6);     System.out.println(p.getKey());        // Legal.
         p = new Pair(2,3);     System.out.println(p.getNumerator());  // Illegal.
Fraction r = new Fraction(4,6); System.out.println(r.getNumerator());  // Legal.
Rational q = new Rational(6,3); System.out.println(q.getNumerator());  // Legal.
```

Subtype Polymorphism and Dynamic Method Dispatch

`Object`
`|`
`Pair`
`|`
`Fraction`
`|`
`Rational`



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.

ArrayList

Motivation: Recall that in Chapter 6 we showed how to maintain a dynamically changing collection of integers in a data structure consisting of an array `A` and an integer `size`:

We implemented each of the operations **add**, **remove**, **membership**, **multiplicity**, and **enumeration** with small code patterns. A ready facility with such patterns is important, but we remarked that writing such code directly in your program also has drawbacks:

- The collection has no single name, and thus it is not easily manipulated as one thing.
- The collection's implementation details are not hidden, and thus your program can both break the data structure's representation invariant and come to excessively depend on its details.

We address those limitations now by defining class `ArrayList`:

- References to instances of `ArrayList` can be manipulated as one thing, i.e., as objects.
- The details of an `ArrayList` are hidden using the class's visibility mechanism, which allows easy replacement of one collection implementation with another.

Guide: The implementation turns the familiar code fragments of Chapter 12 into methods, and will therefore need little additional explanation.

There is an additional benefit of turning these code fragments into the methods that was not previously mentioned:

- The data structure (and its methods) can be instantiated multiple times, i.e., you can easily have as many list objects as you want.

Each method has a header comment that provides the method’s specification. Although various organizations standardize formats for such specifications, we will be less formal about their structure. Nonetheless, we aim for precision that is intended to be adequate external documentation for any client of the class and its interface.

Occasional additional notes are provided, but you should otherwise let the specifications and their implementations speak for themselves.

☞ **Repeatedly improve comments by relentless copy editing.**

ArrayList

Class definition:

```
/* A list of unbounded capacity containing items of type int. */
class ArrayList {
   /* Representation. */
      private int[] A;   // A[0..size-1] is a collection of list items of type int.
      private int size;  // size is the current number of items in the list.
                         // The current list capacity is A.length.
```

ArrayList

The type of an `ArrayList` element is **`int`** (for now), and will be colored blue.

Class definition:

```
s/* A list of unbounded capacity containing items of type int. */
class ArrayList {
   /* Representation. */
      private int[] A;   // A[0..size-1] is a collection of list items of type int.
      private int size;  // size is the current number of items in the list.
                         // The current list capacity is A.length.
```

Integers unrelated to the type of `ArrayList` elements will *not* be colored blue.

Data representation is *private*, i.e., hidden to clients.

ArrayList

Class definition:

```
/* A list of unbounded capacity containing items of type int. */
class ArrayList {
   /* Representation. */
      private int[] A;   // A[0..size-1] is a collection of list items of type int.
      private int size;  // size is the current number of items in the list.
                         // The current list capacity is A.length.
```

Two overloaded constructors: One for a specific initial capacity, the other for a default capacity.

ArrayList

Class definition:

```
/* A list of unbounded capacity containing items of type int. */
class ArrayList {
   /* Representation. */
      private int[] A;   // A[0..size-1] is a collection of list items of type int.
      private int size;  // size is the current number of items in the list.
                         // The current list capacity is A.length.

   /* Constructors. */
.      /* Construct an empty list for int items, with an initial capacity m>=0.
          Throw a ValueError exception if m<0. */
      public ArrayList( int m ) {
         if ( m<0 ) throw new IllegalArgumentException();
         A = new int[m];
         }

      /* Construct an empty list for int items, with an initial capacity of 20. */
      public ArrayList() { this( 20 ); }
```

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

```
...

/* Size. */

  /* Return the number of items in the list. */
  public int size() { return size; }

  /* Return true iff the list is empty. */
  public boolean isEmpty() { return size==0; }
```

ArrayList

```
...

/* Access. */

   /* Return the list item at index k.
      Throw IndexOutOfBoundsException for an out-of-bounds k. */
   public int get(int k) {
      checkBoundExclusive(k);
      return A[k];
      }

   /* Overwrite the list item at index k with v, and return the old item
        that was there.
     Throw IndexOutOfBoundsException for an out-of-bounds k. */
   public int set(int k, int v) {
      checkBoundExclusive(k);
      int old = A[k];
      A[k] = v;
      return old;
      }
```

ArrayList

Raise exception if k is outside the bounds of the current list, excluding the index of the next available slot.

```
...

/* Access. */

   /* Return the list item at index k.
      Throw IndexOutOfBoundsException for an out-of-bounds k. */
   public int get(int k) {
      checkBoundExclusive(k);
      return A[k];
      }

   /* Overwrite the list item at index k with v, and return the old item
        that was there.
     Throw IndexOutOfBoundsException for an out-of-bounds k. */
   public int set(int k, int v) {
      checkBoundExclusive(k);
      int old = A[k];
      A[k] = v;
      return old;
      }
```

ArrayList

```
...

/* Insertion / Deletion. */

   /* Right-shift items with indices k thru the end of the list (if any) one place,
      and insert v at index k. Increase the list capacity, if necessary.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   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++;
      }

   /* Append v at the end of the list. Increase the list capacity, if necessary. */
   public void add(int v) { add(size, v); }
```

ArrayList

```
...

/* Insertion / Deletion. */

   /* Right-shift items with indices k thru the end of the list (if any) one place,
      and insert v at index k. Increase the list capacity, if necessary.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   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++;
      }

   /* Append v at the end of the list. Increase the list capacity, if necessary. */
   public void add(int v) { add(size, v); }
```

ArrayList

```
...

/* Insertion / Deletion. */

   /* Right-shift items with indices k thru the end of the list (if any) one place,
      and insert v at index k. Increase the list capacity, if necessary.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   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++;
      }

   /* Append v at the end of the list. Increase the list capacity, if necessary. */
   public void add(int v) { add(size, v); }
```

ArrayList

```
...

/* Insertion / Deletion. */

   /* Right-shift items with indices k thru the end of the list (if any) one place,
      and insert v at index k. Increase the list capacity, if necessary.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   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++;
      }

   /* Append v at the end of the list. Increase the list capacity, if necessary. */
   public void add(int v) { add(size, v); }
```

ArrayList

Raise IndexError exception if k is outside the bounds of the current list, but allow the index of the next available slot.

```
...

/* Insertion / Deletion. */

   /* Right-shift items with indices k thru the end of the list (if any) one place,
      and insert v at index k. Increase the list capacity, if necessary.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   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++;
      }

   /* Append v at the end of the list. Increase the list capacity, if necessary. */
   public void add(int v) { add(size, v); }
```

ArrayList

```
...

/* Insertion / Deletion. */
   ...
   /* Return the list item with index k after left-shifting items with indices
      k+1 thru the end (if any) to remove the old k-th value from the list.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   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;
      }

   /* Return false if v is not in the list, else remove (one copy of) v from list
      and return true. */
   public boolean removeByValue(int v) {
     int k = indexOf(v);
     if (k == -1) return false; else { remove(k); return true; }
     }
```

ArrayList

```
...

/* Capacity. */

   /* Increase the list's capacity to the maximum of min_capacity or double its
      current 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;
         }
      }
```

ArrayList

```
...

/* Membership. */

   /* Return the index of an instance of v in the list, or -1 if there are none. */
   public int indexOf(int v) {
      int k = 0; while ( (k<size) && (v!=A[k]) ) k++;
      if ( k==n ) return -1; else return k;
      }

   /* Return true iff the list contains (one or more copies of) v. */
   public boolean contains(int v) {
      return indexOf(v)!=-1;
      }
```

```
...

/* Bounds Checking. */

   /* Throw IndexOutOfBoundsException if k is not the index of one of the list’s
      items. */
   private void checkBoundExclusive( int k ) {
      if (k<0 || k>=size) throw new IndexOutOfBoundsException( "≥size" );
      }

   /* Throw IndexOutOfBoundsException if k is not the index of one of the list's items,
     or the next available index for an item to be added. */
   private void checkBoundInclusive( int k ) {
      if (k<0 || k>size) throw new IndexOutOfBoundsException( ">size" );
      }

 } /* ArrayList */
```

ArrayList

Unit Test: Cover every public aspect of the class’s interface (*black-box* testing), and if you know the implementation internals, every corner case you can foresee (*white-box* testing).

```
/* Two useful shorthand functions for the tests that follow. */
private void print(String s) { System.out.println(s); }
private void diag() {
    print("size:"        + " " + collection.size());
    print("isEmpty:"     + " " + collection.isEmpty());
    print("contains 10:" + " " + collection.contains(10));
    print("contains 20:" + " " + collection.contains(20));
    print("index of 10:" + " " + collection.indexOf(10));
    print("index of 20:" + " " + collection.indexOf(20));
    print("-------")
    }
```

☞ **Validate output thoroughly.**

Testing

Test code

```
ArrayList collection;
collection = new ArrayList();
print("new array list:");
diag();
```

Output

```
new array list:
size: 0
isEmpty: true
contains 10: false
contains 20: false
index of 10: -1
index of 20: -1
-------
```

```
collection.add(10);
print("add 10");
diag();
```

```
add 10
size: 1
isEmpty: false
contains 10: true
contains 20: false
index of 10: 0
index of 20: -1
-------
```

```
collection.add(20);
print("add 20");
diag();
```

```
add 20
size: 2
isEmpty: false
contains 10: true
contains 20: true
index of 10: 0
index of 20: 1
-------
```

```
collection.removeByValue(10);
print("remove by value 10");
diag();
```

```
remove by value 10
size: 1
isEmpty: false
contains 10: false
contains 20: true
index of 10: -1
index of 20: 0
-------
```

Test code (continued)

```
collection.add(0,10);
print("add 10 at index 0");
diag();
```

Output (continued)

```
add 10 at index 0
size: 2
isEmpty: false
contains 10: true
contains 20: true
index of 10: 0
index of 20: 1
-------
```

```
collection.add(1,15);
print("add 15 at index 1");
diag();
```

```
add 15 at index 1
size: 3
isEmpty: false
contains 10: true
contains 20: true
index of 10: 0
index of 20: 2
-------
```

```
v = collection.get(1);
print("item at 1" + " " + v);
diag();
```

```
item at 1 15
size: 3
isEmpty: false
contains 10: true
contains 20: true
index of 10: 0
index of 20: 2
-------
```

```
v = collection.set(1, 16);
print("set:" + " " + v + " " +
      "at 1 to 16");
diag();
```

```
set: 15 at 1 to 16
size: 3
isEmpty: false
contains 10: true
contains 20: true
index of 10: 0
index of 20: 2
-------
```

Test code	Output

`v = collection.get(1);`
`print("item at 1 is:" + " " + v);`
`diag();`
	`item at 1 is: 16`
	`size: 3`
	`isEmpty: false`
	`contains 10: true`
	`contains 20: true`
	`index of 10: 0`
	`index of 20: 2`
	`-------`

`collection.add(10);`
`print("add 10");`
`diag();`
	`add 10`
	`size: 1`
	`isEmpty: false`
	`contains 10: true`
	`contains 20: false`
	`index of 10: 0`
	`index of 20: -1`
	`-------`

Unit Test: Seemingly mindless, but surprisingly effective. The skill involves ferreting out every way in which the code might fail.

(1) Exercise every line of code to make sure it does not trigger a crash.

(2) Visually inspect the output to confirm that it is correct.

Can you think of any cases we have missed?

Test code

```
v = collection.get(1);
print("item at 1 is:" + " " + v);
diag();
```

Output

```
item at 1 is: 16
size: 3
isEmpty: false
contains 10: true
contains 20: true
index of 10: 0
index of 20: 2
-------
```

```
collection.add(10);
print("add 10");
diag();
```

```
add 10
size: 1
isEmpty: false
contains 10: true
contains 20: false
index of 10: 0
index of 20: -1
-------
```

Unit Test: Seemingly mindless, but surprisingly effective. The skill involves ferreting out every way in which the code might fail.

(1) Exercise every line of code to make sure it does not trigger a crash.

(2) Visually inspect the output to confirm that it is correct.

Can you think of any cases we have missed?

Visual inspection of output is tedious, and not something you want to redo manually after every code change. It is common to automate such retests by capturing the desired output in a file to which new output can be compared automatically after each change.

Enumeration of Rationals, continued

Enumeration of rationals: Recall this incomplete code example from Chapter 6.

```
/* 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 ∪ {z}; */
            }
         r--;
         }
      d++;
      }
```

Enumeration of Rationals, continued

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

```
/* 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++;
      }
```

Enumeration of Rationals, continued

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.

```
/* 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++;
      }
```

Parametric Polymorphism and Generic Classes

Enumeration of Rationals, continued

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 class definition that is parametrized by a type is called a *generic class*.

Parametric Polymorphism and Generic Classes

ArrayList<E>

The type of an `ArrayList` item is parameterized as **E**.

Generic class definition:

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

```
/* A list of unbounded capacity containing items of type E. */
class ArrayList <E> {
   /* Representation. */
      private E[] A;     // A[0..size-1] is a collection of list items of type E.
      private int size;  // size is the current number of items in the list.
                         // The current list capacity is A.length.

   /* Constructors. */
      /* Construct an empty list for E items, with an initial capacity m>=0.
         Throw a IllegalArgumentException if m<0. */
      public ArrayList( int m ) {
         if ( m<0 ) throw new IllegalArgumentException();
         A = (E[]) new Object[m];
         }

      /* Construct an empty list for E items, with an initial capacity of 20. */
      public ArrayList() { this( 20 ); }
```

Parametric Polymorphism and Generic Classes

ArrayList<E>

We will not repeat the definitions of every method, but will let these two illustrate what is needed. Essentially, every (blue) **int** is turned into a type parameter **E**.

```
...

/* Access. */

   /* Return the list item at index k.
      Throw IndexOutOfBoundsException for an out-of-bounds k. */
   public E get(int k) {
      checkBoundExclusive(k);
      return A[k];
      }

   /* Overwrite the list item at index k with v, and return the old item
        that was there.
     Throw IndexOutOfBoundsException for an out-of-bounds k. */
   public E set(int k, E v) {
      checkBoundExclusive(k);
      E old = A[k];
      A[k] = v;
      return old;
      }
```

Parametric Polymorphism and Generic Classes

ArrayList<E>

A non-obvious subtlety in method `remove` involves an erasure step that assists in the efficient management of storage. This is explained in the Garbage Collection discussion, later.

```
...
/* Insertion / Deletion. */
   ...
   /* Return the list item with index k after left-shifting items with indices
      k+1 thru the end (if any) to remove the old k-th value from the list.
      Throw IndexOutOfBoundsException on out-of-bounds k. */
   public int remove(int k) {
      checkBoundExclusive(k);
      int old = A[k];
      size--;
      for (int j=k; j<size; j++) A[j] = A[j+1];
      A[size] = null; // Garbage-collection assist.
      return old;
      }

   /* Return false if v is not in the list, else remove (one copy of) v from list
      and return true. */
   public boolean removeByValue(E v) {
     int k = indexOf(v);
     if (k != -1) { remove(k); return true; }
     else return false;
     }
```

Parametric Polymorphism and Generic Classes

`ArrayList<E>`

Parameters of `indexOf` and `contains` generalized to take any `Object`, and search changed to use the `equals` operation of the argument `v` rather than `==`.

```
...
/* Membership. */

   /* Return the index of an instance of v in the list, or -1 if there
      are none. */
   public int indexOf(Object v) {
      int k = 0; while ( (k<n) && (!v.equals(A[k])) ) k++;
      if ( k==n ) return -1; else return k;
      }
   /* Return true iff the list contains (one or more copies of) v. */
   public boolean contains(Object v) { return indexOf(v)!=-1; }
```

Enumeration of Rationals, continued

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++;
    }
```

Enumeration of Rationals, continued

Enumeration of rationals: We declare `reduced` to have type `ArrayList<Rational>`.

```
/* 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++;
      }
```

Parametric Polymorphism and Generic Classes

Enumeration of Rationals, continued

Technically, the generic constructor `ArrayList<E>()` is being instantiated as `ArrayList<Rational>()`, which Java will do for you behind the scene.

Enumeration of rationals: We declare `reduced` to have type `ArrayList<Rational>`.

```
/* 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, continued

Enumeration of rationals: and obtain the correct output.

```
1
2
1/2
3
     ⟵ 2/2 omitted
1/3
4
3/2
2/3
1/4
5
     ⟵ 4/2, 3/3, and 2/4 omitted
1/5
6
5/2
4/3
3/4
2/5
1/6
7
     ⟵ 6/2 omitted
5/3
     ⟵ 4/4 omitted
3/5
     ⟵ 2/6 omitted
1/7
etc.
```

Parametric Polymorphism and Generic Classes

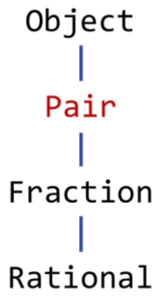
Enumeration of Rationals, continued

Enumeration of rationals: and obtain the correct output.

1
2
1/2
3 ⟵ 2/2 omitted
1/3
4
3/2
2/3
1/4
5 ⟵ 4/2, 3/3, and 2/4 omitted
1/5
6
5/2
4/3
3/4
2/5
1/6
7 ⟵ 6/2 omitted
5/3 ⟵ 4/4 omitted
3/5 ⟵ 2/6 omitted
1/7
etc.

Generic Classes

Pair<K,V>


Object

Pair

Fraction

Rational

Class definition: Recall the definition of class `Pair`.

```
class Pair {
   /* Representation. */
      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 Classes

Pair<K,V>

Generic class definition: It, too, can be made generic so we can have pairs of any types.

```
class Pair<K,V> {
   /* Representation. */
      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 Classes

`Pair<K,V>`

Generic class definition: It, too, can be 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 `==`.

Generic Classes

`Pair<K,V>`

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;
```

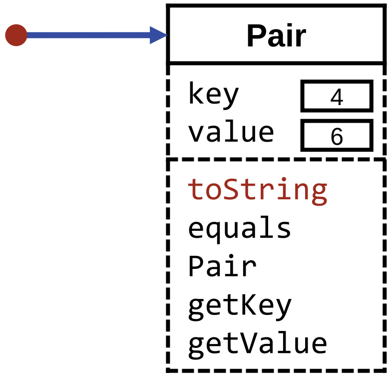
because **`int`** is a *primitive type*, not an object type.

We deal with this next.

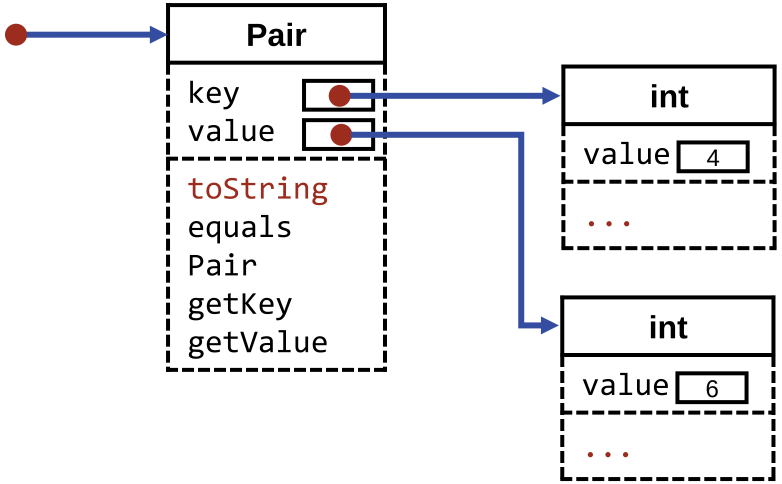
Uniformity:

- In some languages, e.g., Python, all values are uniformly objects of some class, and each value is accessed via a reference.
- The object reference (●) has a standard size, but the object itself doesn't.
- In such languages, even values of basic types like **`int`** and **`bool`** are objects.

Simplified depiction



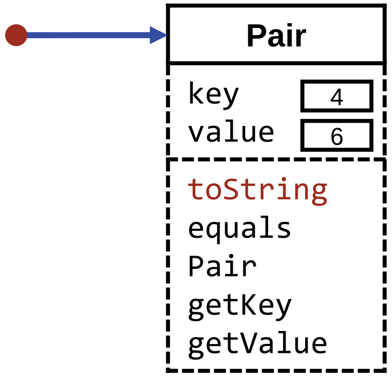
More accurate depiction



Uniformity:

- Other languages, e.g., Java, distinguish between *primitive values* and objects of a class.
- Primitive values, e.g., values of types **`int`** and **`boolean`**, fit conveniently into variables of standard sizes, and are not accessed via a reference.
- In such languages, the depiction characterized as "simplified" is actually accurate.

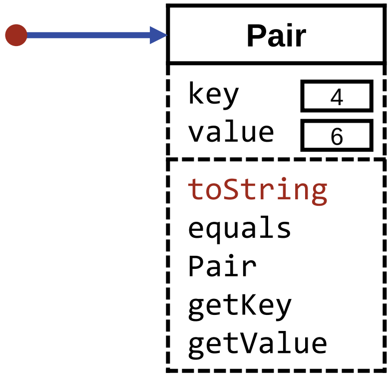
Accurate depiction



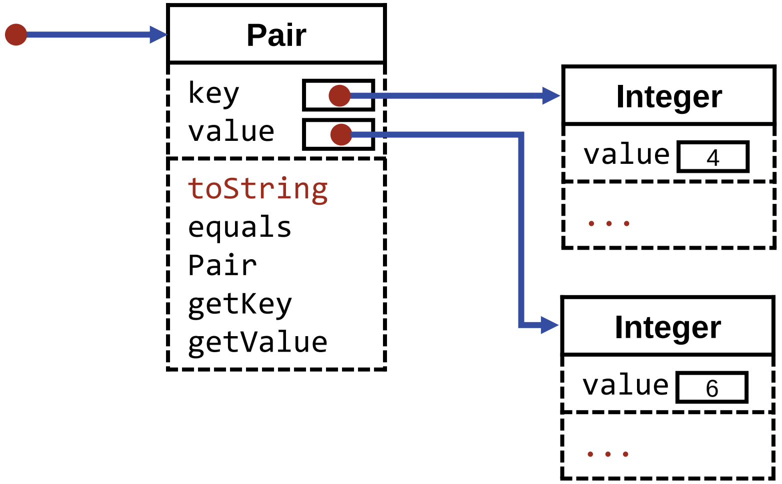
Uniformity:

- In the interest of efficiency, but at the expense of complexity, Java offers two worlds, one in which values of types like **`int`** and **`boolean`** are *primitive*, and the other in which there are *object versions* of such values (of types `Integer` and `Boolean`) known as *boxed* integers and Booleans.
- Crossing back and forth between the two worlds is a bit complicated, but is ameliorated by features known as *auto-boxing* and *auto-unboxing*.

Accurate depiction



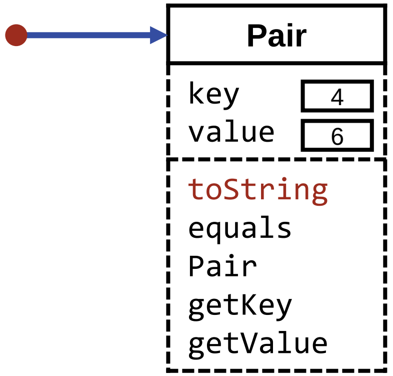
Accurate depiction



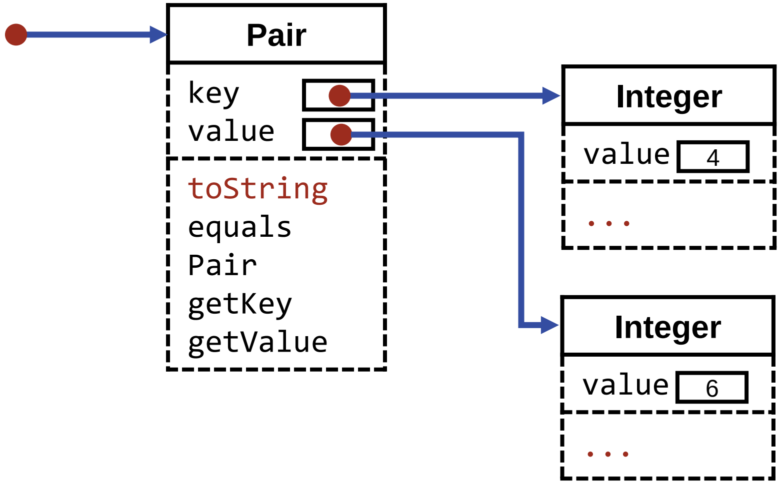
Uniformity:

- An advantage of a language in which all values are objects is that generic classes can be instantiated with any types. In contrast, in a language that distinguished between primitive values and objects, generic classes can not be instantiated with primitive types such as **`int`** and **`boolean`**.

Accurate depiction

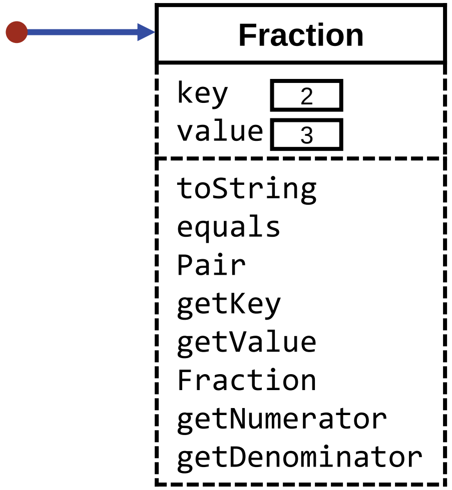


Accurate depiction



Fraction

Object
|
Pair
|
Fraction
|
Rational

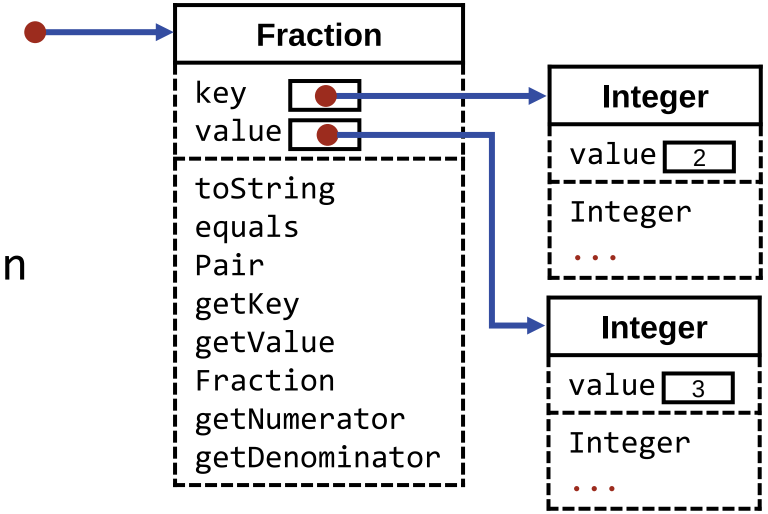


Subclass definition: Recall the definition of `Fraction`.

```
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 */
```

Fraction

```
      Object
        |
Pair<Integer,Integer>
        |
     Fraction
        |
     Rational
```

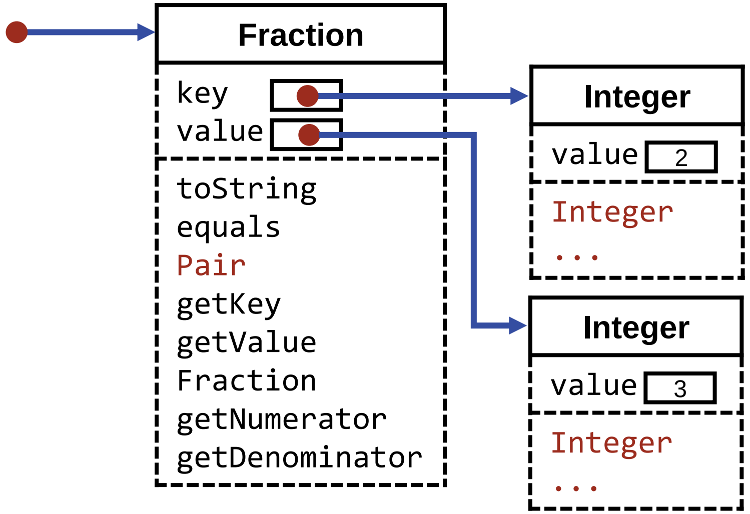


Subclass definition: Since `Pair` is now a generic class, Fraction must instantiate it with component types.

```
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 */
```

Fraction

Object
|
Pair<Integer,Integer>
|
Fraction
|
Rational



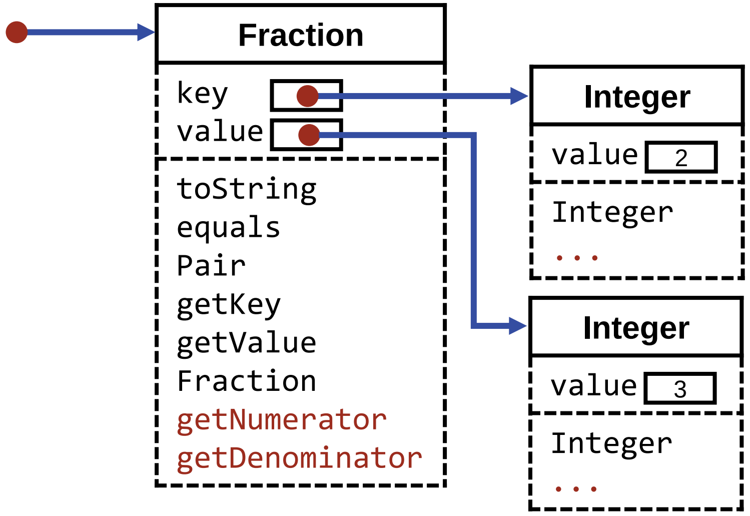
Subclass definition: Auto-boxing and auto-unboxing occurs between `int` values and `Integer` values.

```
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, which now expects `Integer` arguments.

Fraction

Object
Pair<Integer,Integer>
Fraction
Rational



Fraction
key
value
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator

Integer
value 2
Integer
...

Integer
value 3
Integer
...

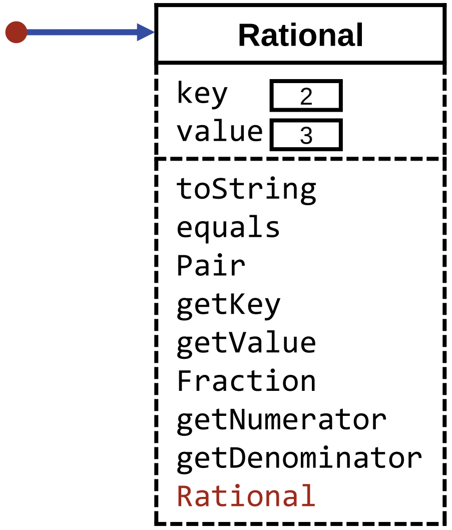
Subclass definition: Auto-boxing and unboxing occurs between **`int`** values and `Integer` values.

```
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 `key` and `value` fields (which are now type `Integer`) occurs when they returned as the values of the getters, which are expected to return values of type **`int`**.

Rational

Object
Pair
Fraction
Rational

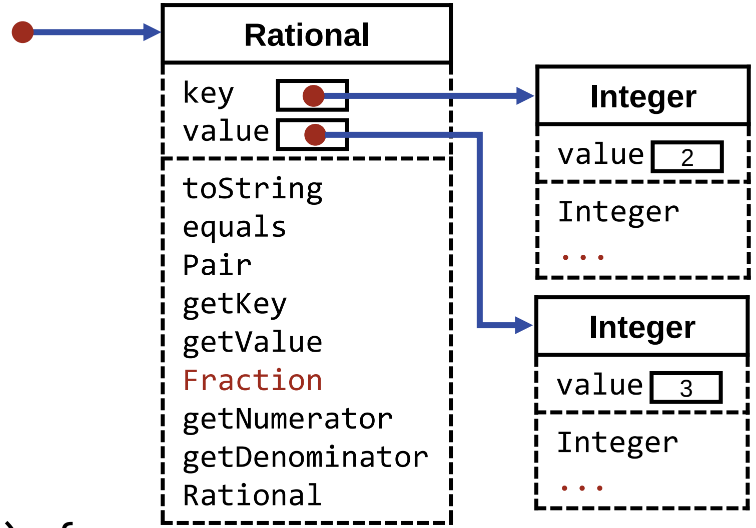

Rational
key 2
value 3
toString
equals
Pair
getKey
getValue
Fraction
getNumerator
getDenominator
Rational

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 */
```

Rational

Object
|
Pair<Integer,Integer>
|
Fraction
|
Rational



Subclass definition: Since `Fraction` inherits from `Pair<Integer,Integer>`, so too does `Rational`.

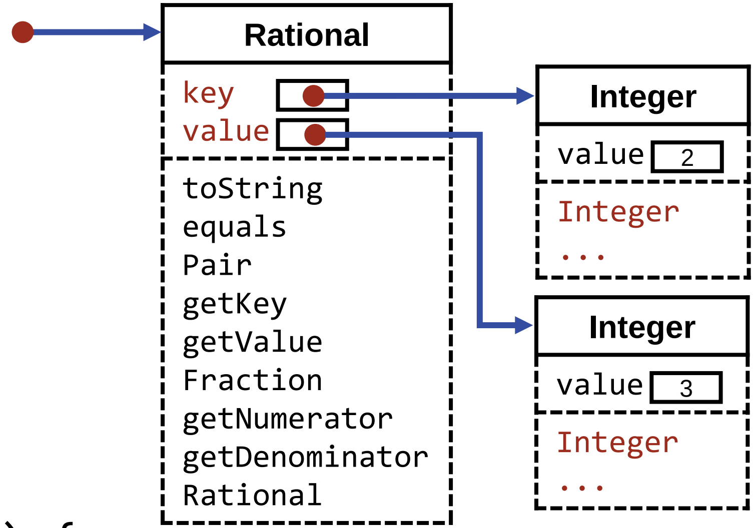
```
class Rational extends Fraction {
   /* Constructor */
      public Rational(int numerator, int denominator) {
         super(numerator, denominator);    // Apply the Fraction constructor.
         int g = gcd(numerator, denominator);
         key = numerator/g;
         value = denominator/g;
 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

No auto-boxing of `int` parameters `numerator` and `denominator` occurs when they are passed to the `Fraction` constructor because it expects two `int` arguments. But they are auto-boxed when it invokes the `Pair` constructor.

Rational

Object
Pair<Integer,Integer>
Fraction
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 */
```

Auto-boxing of the computed `int` values `numerator/g` and `denominator/g` occurs when they are assigned to the `key` and `value` fields (which are now type `Integer`).

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.

e.g., the code executed for `toString` depends on the type of the object, e.g., `Rational`.

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>` or `Pair<K,V>`.

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<Rational>` or `Pair<Integer,Integer>`.

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.

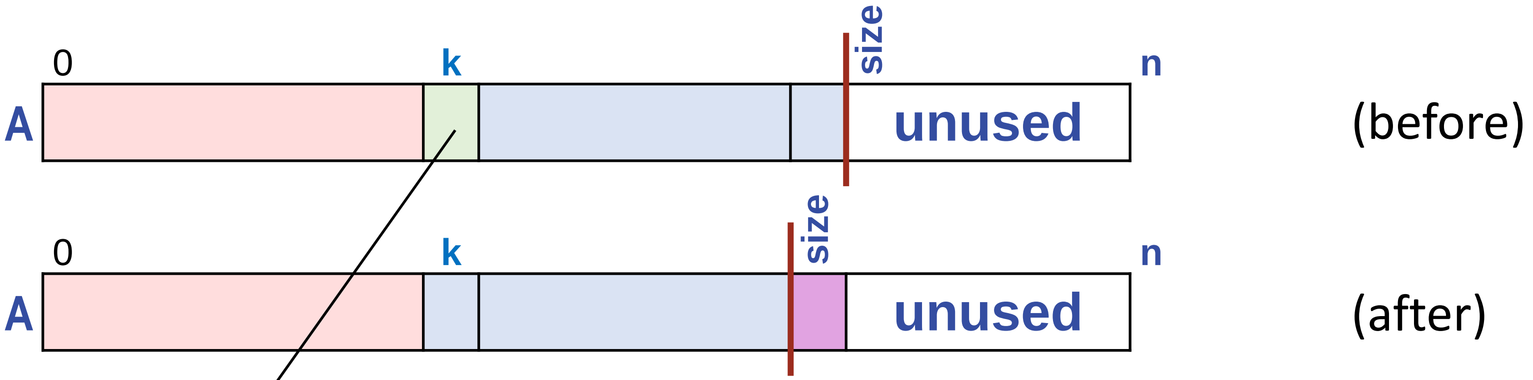
Polymorphism: Four kinds have been illustrated.

- **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.

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

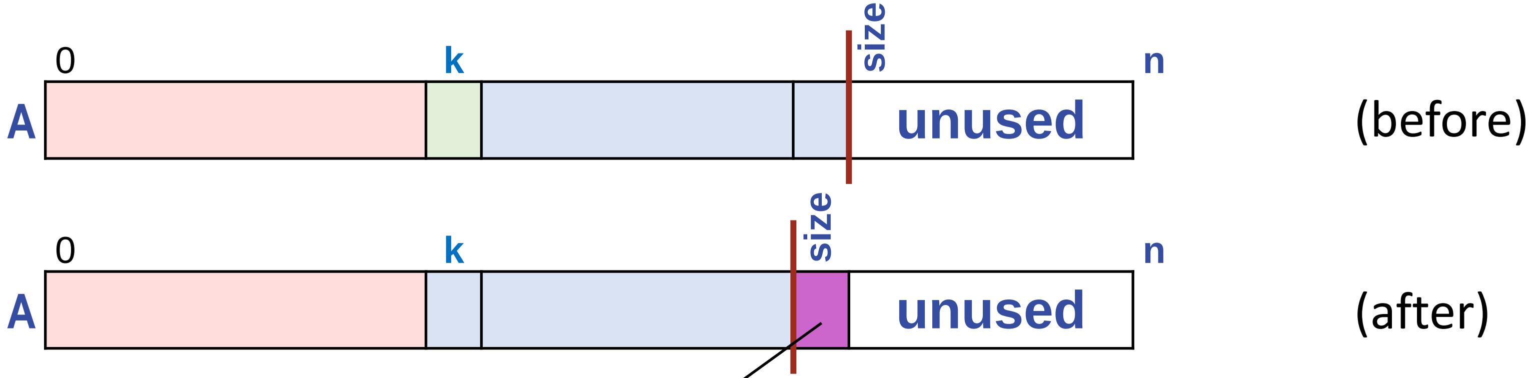


Garbage Collection. Recall the definition of `remove` in `ArrayList[E]`.

```
public E remove(int k) {
         checkBoundExclusive(k);
         E old = A[k];
         size--;
         for (int j=k; j<size; j++) A[j] = A[j+1];
         A[size] = null; // Garbage-collection assist.
         return old;
         }
```

The left shift of (blue) values overwrites the (green) k-th value that is being removed from the collection. It was a reference to some object of arbitrary size and complexity, and if this had been the only reference to that object, it could now be garbage collected. But the value being removed at `A[k]` is not the issue, as that reference is being overwritten.

Garbage Collection



Garbage Collection. Recall the definition of `remove` in `ArrayList[E]`.

```
public E remove(int k) {
        checkBoundExclusive(k);
        E old = A[k];
        size--;
        for (int j=k; j<size; j++) A[j] = A[j+1];
        A[size] = null; // Garbage-collection assist.
        return old;
        }
```

The issue is the *last* (blue) value in the collection, which was originally in `A[size-1]`, and that has now been left-shifted one place. A copy of that value remains in `A[size]`, the first element of the unused array suffix. It is that violet copy that we must nullify. Note that the object referred to by the violet reference can not yet be collected because a reference to it remains in `A[size-1]`. However, if and when *that* reference is removed or is overwritten, the object in question will *then* be collectable by virtue of our having nullified the problematic copy in `A[size]`.

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.

Alternative Implementations

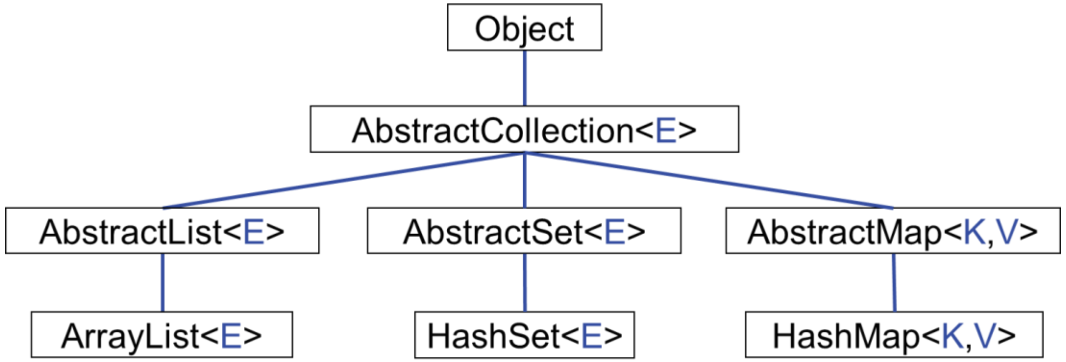
Abstract Data Types

A class that hides all of its implementation details, and only exposes its public methods is known as an *abstract data type*. The names, return types, and parameter types are known as the class's *interface*.

Writing code with abstract data types permits the (relatively easy) replacement of one implementation with an another, a decided advantage.

We illustrate this by using `HashSet[E]`, an alternative to `ArrayList[E]` that can be found in the Library

Libraries: The library `java.util` contains many useful classes, including these for collections:



Object
AbstractCollection<E>
AbstractList<E>
AbstractSet<E>
AbstractMap<K,V>
ArrayList<E>
HashSet<E>
HashMap<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.

HashSet

Enumeration of Rationals, continued

Enumeration of rationals: Recall our code for enumerating rationals using `ArrayList<E>`.

```
/* 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++;
      }
```

HashSet

Enumeration of Rationals, continued

Enumeration of rationals: To use `HashSet<E>` instead, we only need to change one word.

```
/* 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++;
      }
```

The text of the `contains` and `add` invocations is unchanged, but the methods that are actually invoked change radically, i.e., from the `ArrayList<E>` implementations to the `HashSet<E>` implementations.

HashSet

Enumeration of Rationals, continued

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.

HashSet

Enumeration of Rationals, continued

Enumeration of rationals: Contrast performance of `ArrayList` and `HashSet`.

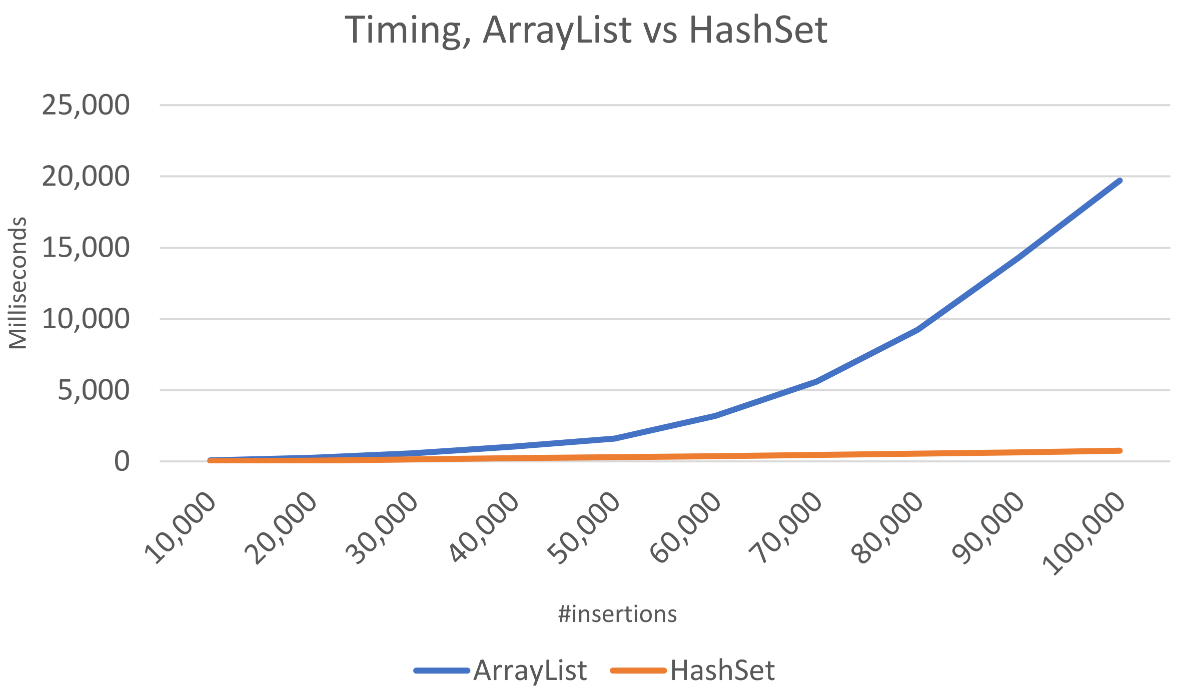
```
/* Output reduced fractions, i.e., positive rationals; no repeats. */
public static void timing() {
  HashSet<Rational> reduced = new HashSet<>();
  long startTime = System.currentTimeMillis();
  int rCount = 0;  // # of rationals so far.
  int d = 0;
  while ( rCount<100000 ) {
    int r = d;
    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.

HashSet

Enumeration of Rationals, continued

Enumeration of rationals: performance of `ArrayList` vs. `HashSet`.



Performance of ArrayList is quadratic; performance of HashSet is linear.

HashSet

Enumeration of Rationals, continued

Timing Study: But why are we bothering to maintain the collection of already-output rationals in the first place? We thought this was needed to make sure we would only emit each rational once.

But why not just output the fractions that are in reduced form as they arise? We didn't actually need the collection in the first place. It was all just a pedagogical ruse!

The test for *n*/*d* being in reduced form is "gcd(*n*,*d*)==1".

☞ **Analyze first.**

HashSet

Enumeration of Rationals, continued

Enumeration of rationals: Contrast performance of `ArrayList`, `HashSet`, and gcd=1.

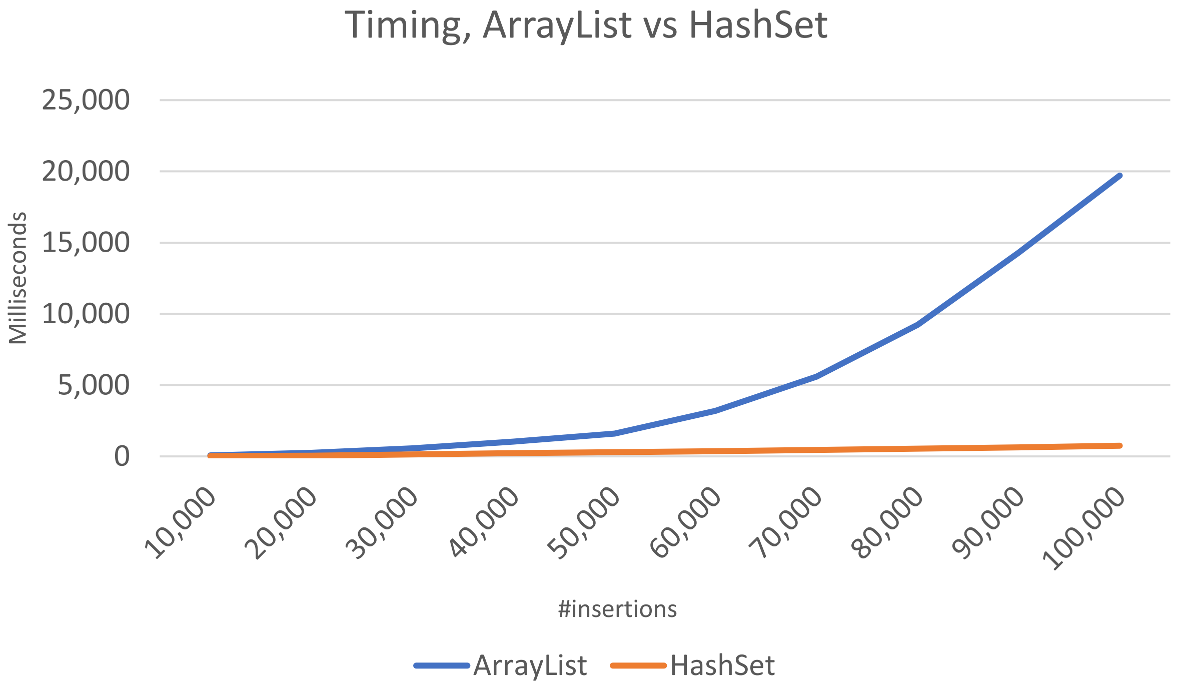
```
/* Output reduced fractions, i.e., positive rationals; no repeats. */
public static void timing() {

   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 */
```

Only create the `Rational` when the fraction is in reduced form.

HashSet

Enumeration of Rationals, continued

Enumeration of rationals: Contrast performance of `ArrayList`, `HashSet`, and gcd=1.



#insertions	ArrayList	HashSet	gcd
10,000	72	23	2
20,000	257	50	5
30,000	574	135	8
40,000	1035	220	11
50,000	1601	308	14
60,000	3206	372	16
70,000	5602	463	18
80,000	9236	550	20
90,000	14290	644	23
100,000	19711	750	27

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.,

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

```
/* 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 `AbstractCollection<E>`. Let `c` be an object of an instantiation of `C<E>` for some specific element type `EL`. Then `c` is a collection of `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 `EL`. Provided `i.hasNext()` would return **true**, invoking `i.next()` would return 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 `Iterator<E>` that has an instantiation `Iterator<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:

```
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:

```
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()`.

Summary:

We have presented the flavor of Object-Oriented Programming (OOP), and some of its technical details.

We reinforced many of the lessons of earlier chapters:

- Combining careful specification of statements, declarations, and methods with careful implementations.
- The practice of incremental testing.
- The benefit of analysis.

Object-Oriented Programming addresses for code many issues that scholars have considered in Philosophy:

- The nature of taxonomy.
- Abstraction and instantiation.
- Ontology and epistemology.