Mutable Classes (cont)

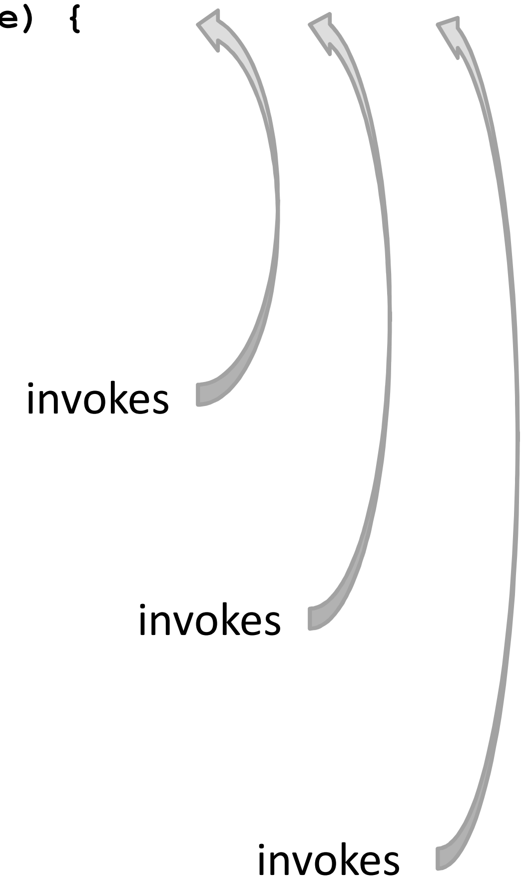
Constructors

```
public Vector2D(double x, double y, String name) {
  this.x = x;
  this.y = y;
  this.name = name;
}

public Vector2D() {
  this(0, 0, null);
}

public Vector2D(double x, double y) {
  this(x, y, null);
}

public Vector2D(Vector2D other) {
  this(other.x, other.y, other.name);
}
```



Constructor Chaining

- when a constructor invokes another constructor it is called *constructor chaining*
- to invoke a constructor in the same class you use the **`this`** keyword
  - if you do this then it must occur on the first line of the constructor body

Accessor Methods

- recall that accessor methods return information about the state of the object
  - for **`Vector2D`** we need to return information about **`x`**, **`y`**, and **`name`**
- we have 3 accessor methods

`double getX()` Get the x coordinate of the vector.
`double getY()` Get the y coordinate of the vector.
`String getName()` Get the name of the vector.

Accessor Methods

```
public double getX() {
  return this.x;
}

public double getY() {
  return this.y;
}

public double getName() {
  return this.name;
}
```

Mutator Methods

- recall that mutator methods allow a client to manipulate the state of the object
  - for **`Vector2D`** we need to allow the client to manipulate **`x`**, **`y`**, and **`name`**

Mutator Methods

- we have 5 mutator methods

`void setX(double x)` Set the x coordinate of the vector.
`void setY(double y)` Set the y coordinate of the vector.
`void setName(String name)` Set the name of the vector.
`void set(double x, double y)` Set the x and y coordinate of the vector
`void set(String name, double x, double y)` Set the name, x, and y coordinate of the vector

`setX()`, `setY()`, and `set()`

```
public void setX(double x) {
  this.x = x;
}

public void setY(double y) {
  this.y = y;
}

public void setName(String name) {
  this.name = name;
}

public void set(double x, double y) {
  this.setX(x);
  this.setY(y);
}

public void set(String name, double x, double y) {
  this.setName(name);
  this.set(x, y);
}
```

Equals

- recall that most value type classes will want their own version of **`equals`**
  - we shall say that two vectors are equal if their **`x`**, and **`y`** coordinates are equal
    - i.e., two vectors might be equal even if their names are different

```
boolean equals(Object obj)
```
Compares two vectors for equality.

equals()

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }




  return eq;
}
```

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass()) {



  }
  return eq;
}
```

```java
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass())  {
    Vector2d other = (Vector2d) obj;


  }
  return eq;
}
```

This version works most of the time (except when it doesn't!)

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass())  {
    Vector2d other = (Vector2d) obj;
    eq = this.getX() == other.getX() &&
         this.getY() == other.getY();
  }
  return eq;
}
```

This version always works.

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass()) {
    Vector2d other = (Vector2d) obj;
    eq = Double.compare(this.getX(), other.getX()) == 0 &&
         Double.compare(this.getY(), other.getY()) == 0;
  }
  return eq;
}
```

== vs Double.compare

- the issue here is quite subtle
- if you use == to compare the coordinates then

```
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`false`** because **`NaN == NaN`** is always **`false`**

- NaN means "not a number" and is used to represent a mathematically undefined number
  - such as occurs when you divide zero by zero
  - the behavior of NaN is defined in the IEEE 754 standard for floating point arithmetic (i.e., this is not just a Java issue)

== vs Double.compare

- if you use == to compare the coordinates then all hash based collections and all sets will behave strangely with vectors having NaN as a component

```
Set<Vector2D> set = new HashSet<Vector2D>();
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
set.add(u);
set.add(v);
System.out.println(set.size());             // prints 2
```

- sets are supposed to reject duplicate elements but there are 2 identical vectors in **`set`**
  - occurs because **`Set`** uses **`equals`** to check for duplicates

== vs Double.compare

- if you use **`Double.compare`** to compare the coordinates then

```
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`true`** because **`Double.compare`** is implemented to allow for equality of **`NaN`**

- checking for equality of **`NaN`** can be useful when trying to track down errors in computations
- also the hash based collections and sets will work as expected

== vs Double.compare

- there is a side effect of using **`Double.compare`** to compare the coordinates

```
Vector2D u = new Vector2D(0.0, 1.0);     // (0.0, 1.0)
Vector2D v = new Vector2D(-0.0, 1.0);    // (-0.0, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`false`** because **`Double.compare`** considers 0.0 and -0.0 to be unequal

  - can you see how to implement equals to allow for equality of **`NaN`** and equality of 0.0 and -0.0?

== vs Double.compare

- the real issue here is that floating point arithmetic is tricky and affects every programming language
- a good starting point for learning more about some of the issues involved
  - http://floating-point-gui.de/

Observe That…

- instead of directly using the fields, we use accessor methods where possible
  - this reduces code duplication, especially if accessing an field requires a lot of code
  - this gives us the possibility to change the representation of the fields in the future
    - as long as we update the accessor methods (but we would have to do that anyway to preserve the API)
    - for example, instead of two attributes `x` and `y`, we might want to use an array or some sort of `Collection`

- the notes [notes 2.3.1] call this *delegating to accessors*

Observe That…

- instead of directly modifying the attributes, we use mutator methods where possible
  - this reduces code duplication, especially if modifying an attribute requires a lot of code
  - this gives us the possibility to change the representation of the attributes in the future
    - as long as we update the mutator methods (but we would have to do that anyway to preserve the API)
  - for example, instead of two attributes **`x`** and **`y`**, we might want to use an array or some sort of **`Collection`**

- the notes [notes 2.3.1] call this *delegating to mutators*

Things to Think About

- how do you implement **`Vector2D`** using an array to store the coordinates?
- how do you implement **`Vector2D`** using a **`Collection`** to store the coordinates?
- how do you implement **`VectorND`**, an N-dimensional vector?

hashCode and compareTo

hashCode()

- if you override **equals()** you must override **hashCode()**
  - otherwise, the hashed containers won't work properly
    - recall that we did not override **hashCode()** for **PhoneNumber**

```
// client code somewhere
PhoneNumber pizza = new PhoneNumber(416, 967, 1111);

HashSet<PhoneNumber> h = new HashSet<PhoneNumber>();
h.add(pizza);
System.out.println( h.contains(pizza) );       // true

PhoneNumber pizzapizza =
                    new PhoneNumber(416, 967, 1111);
System.out.println( h.contains(pizzapizza) );  // false
```


[notes 2.3.5]

Arrays as Containers

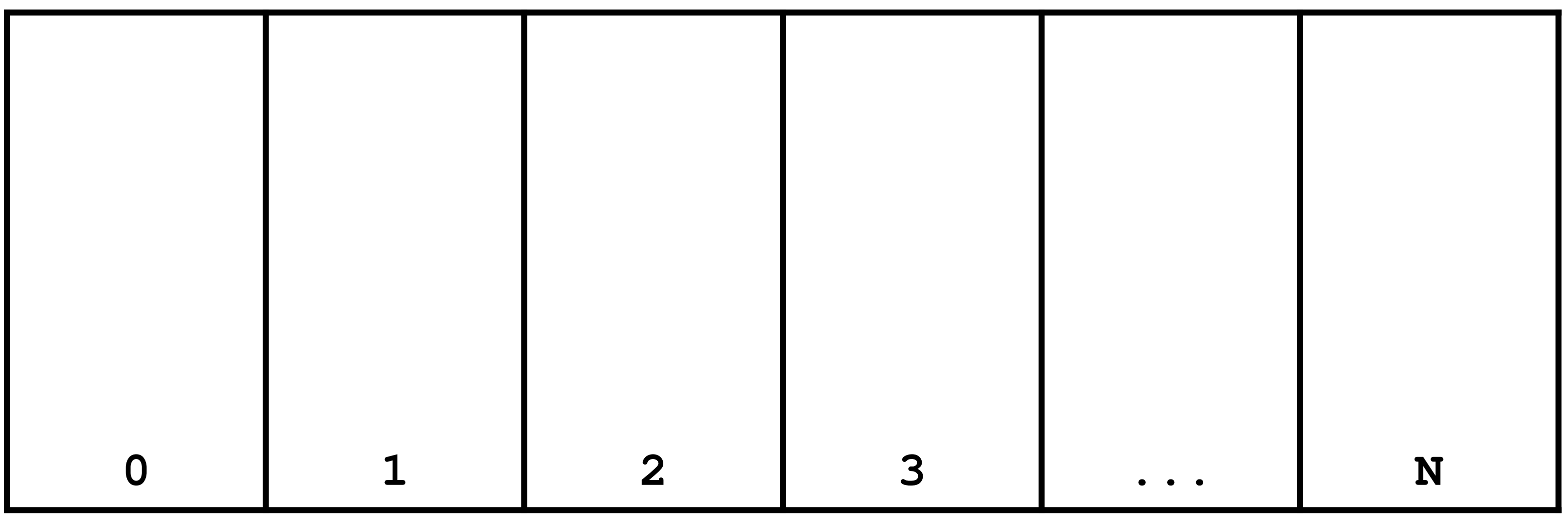
- suppose you have an array of unique **`PhoneNumbers`**
  - how do you compute whether or not the array contains a particular **`PhoneNumber`**?

```
public static boolean
       hasPhoneNumber(PhoneNumber p,
                      PhoneNumber[] numbers)
{
  if (numbers != null) {
    for( PhoneNumber num : numbers ) {
      if (num.equals(p)) {
        return true;
      }
    }
  }
  return false;
}
```

- called *linear search* or *sequential search*
  - doubling the length of the array doubles the amount of searching we need to do
- if there are **`n PhoneNumber`**s in the array:
  - best case
    - the first **`PhoneNumber`** is the one we are searching for
      - 1 call to **`equals()`**
  - worst case
    - the **`PhoneNumber`** is not in the array
      - n calls to **`equals()`**
  - average case
    - the **`PhoneNumber`** is somewhere in the middle of the array
      - approximately (n/2) calls to **`equals()`**

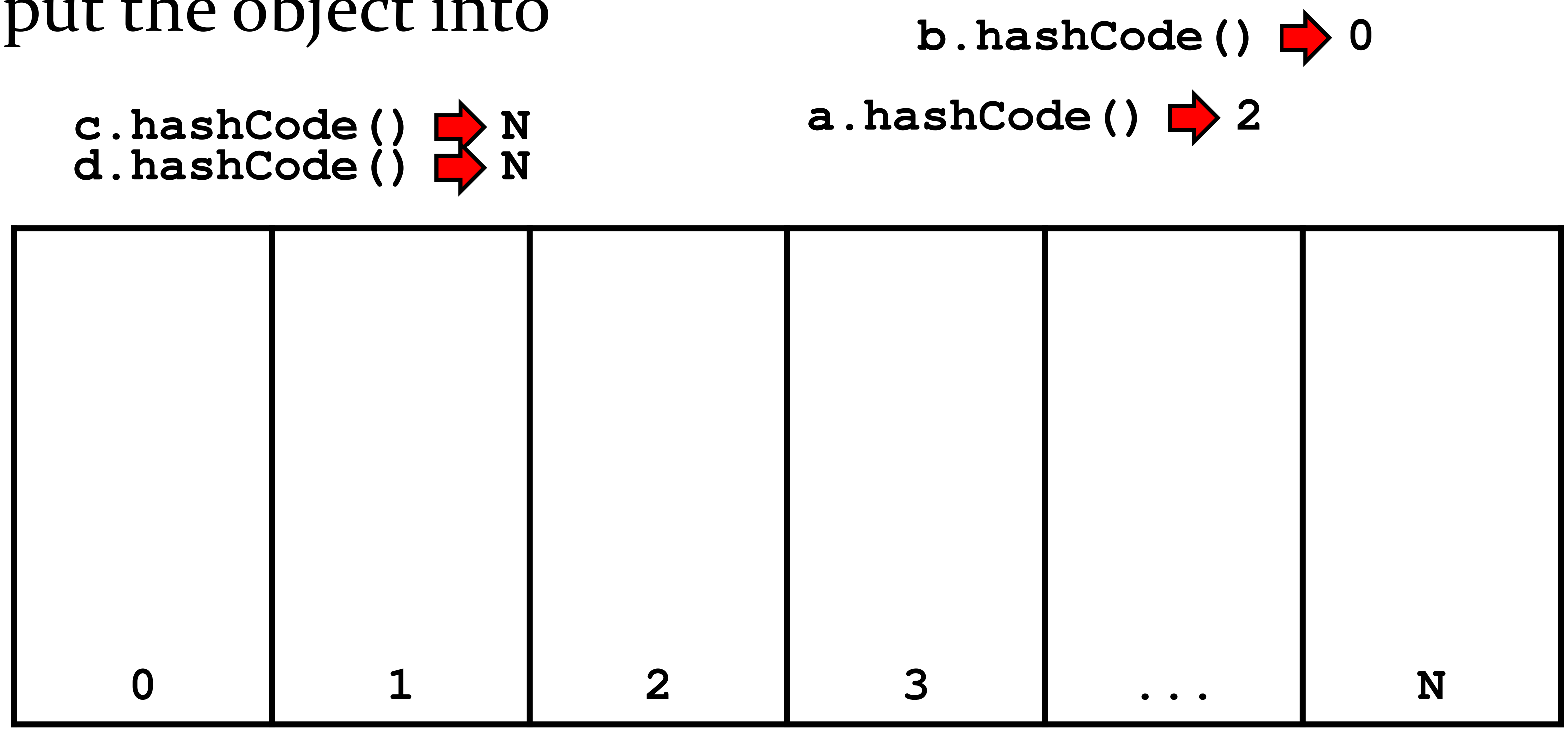
Hash Tables

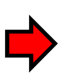
- you can think of a hash table as being an array of buckets where each bucket holds the stored objects



Insertion into a Hash Table

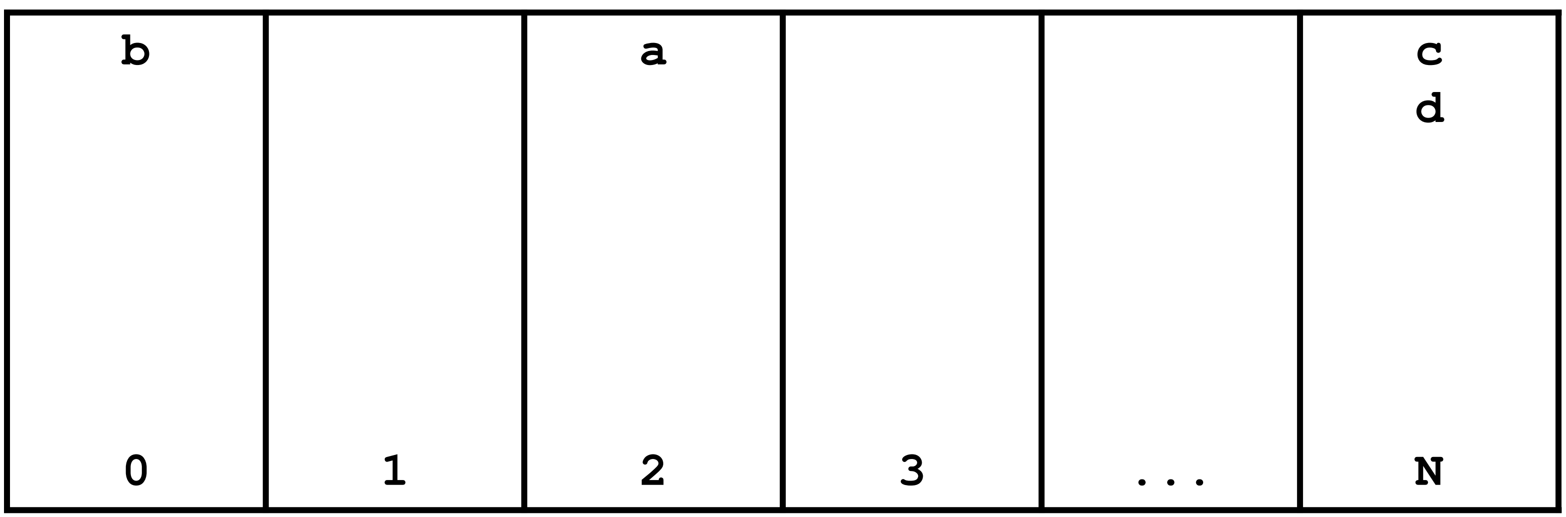
- to insert an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to put the object into



 means the hash table takes the hash code and does something to it to make it fit in the range **`0—N`**

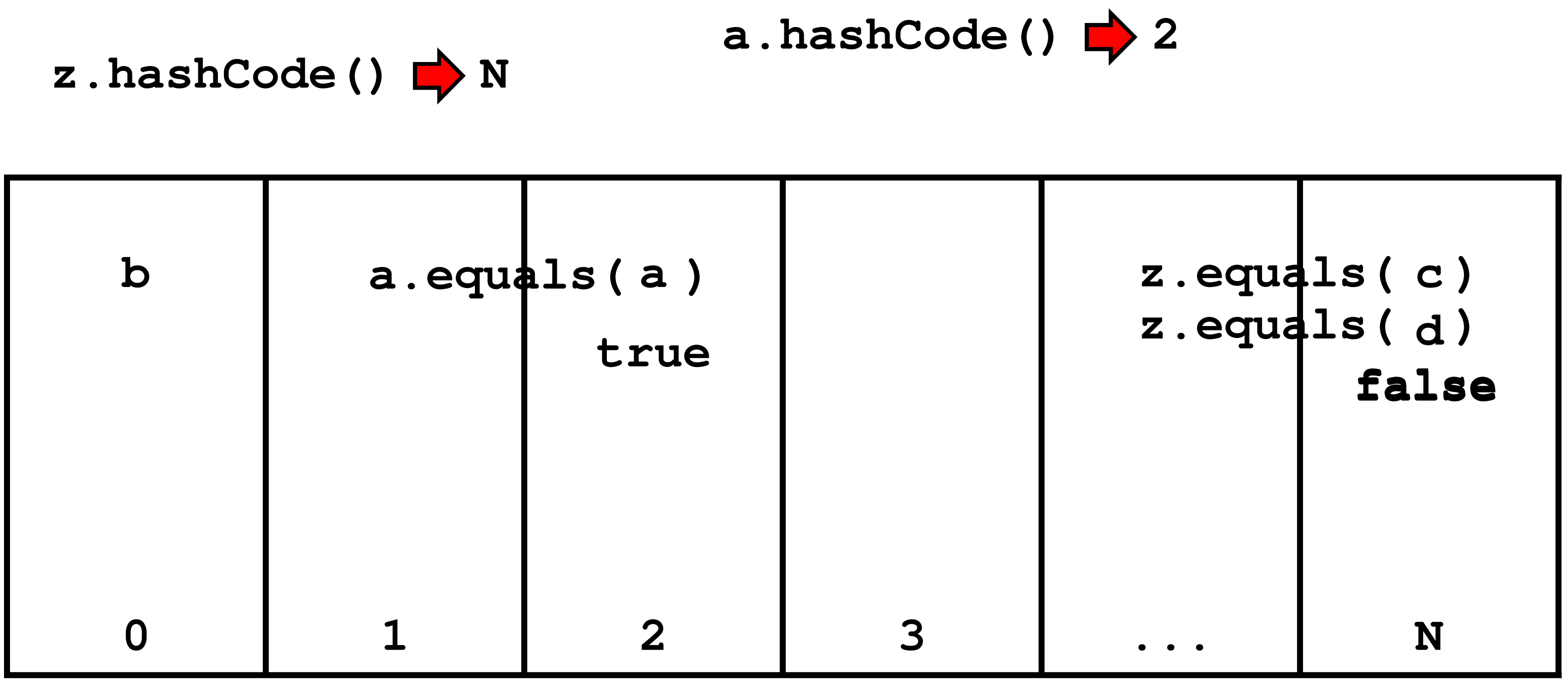
Insertion into a Hash Table

- to insert an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to put the object into



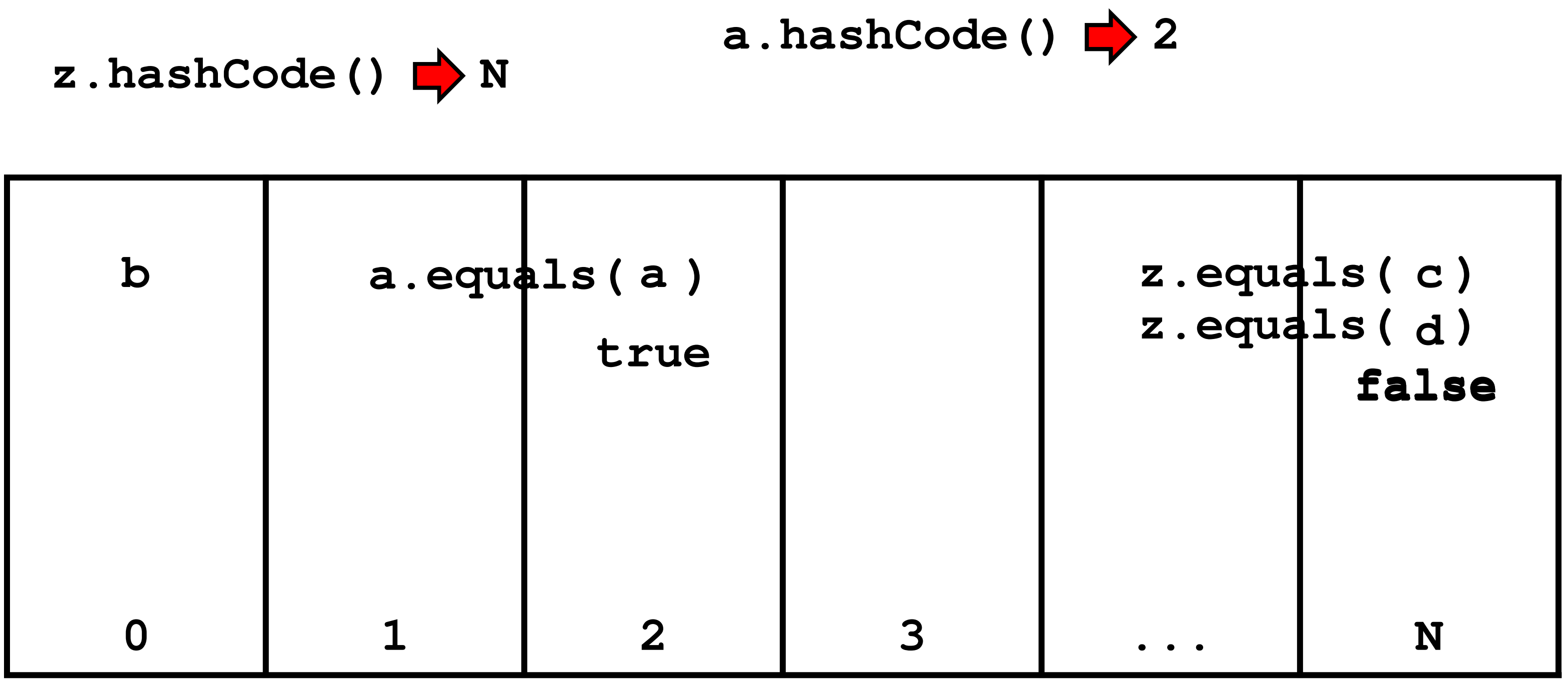
Search on a Hash Table

- to see if a hash table contains an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to look for **`a`** in



Search on a Hash Table

- to see if a hash table contains an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to look for **`a`** in



- searching a hash table is usually much faster than linear search
  - doubling the number of elements in the hash table usually does not noticably increase the amount of search needed
- if there are **`n`** **`PhoneNumber`**s in the hash table:
  - best case
    - the bucket is empty, or the first **`PhoneNumber`** in the bucket is the one we are searching for
      - 0 or 1 call to **`equals()`**
  - worst case
    - all **`n`** of the **`PhoneNumber`**s are in the same bucket
      - n calls to **`equals()`**
  - average case
    - the **`PhoneNumber`** is in a bucket with a small number of other **`PhoneNumbers`**
      - a small number of calls to **`equals()`**

`Object hashCode()`

- if you don't override `hashCode()`, you get the implementation from `Object.hashCode()`
  - `Object.hashCode()` uses the memory address of the object to compute the hash code

```java
// client code somewhere
PhoneNumber pizza = new PhoneNumber(416, 967, 1111);

HashSet<PhoneNumber> h = new HashSet<PhoneNumber>();
h.add(pizza);

PhoneNumber pizzapizza = new PhoneNumber(416, 967, 1111);
System.out.println( h.contains(pizzapizza) );  // false
```

- note that **`pizza`** and **`pizzapizza`** are distinct objects
  - therefore, their memory locations must be different
    - therefore, their hash codes are different (probably)
    - therefore, the hash table looks in the wrong bucket (probably) and does not find the phone number even though **`pizzapizza.equals(pizza)`** *

* unless you're from Naples

A Bad (but legal) **hashCode()**

```
public final class PhoneNumber {
  // attributes, constructors, methods ...

  @Override public int hashCode()
  {
    return 1;  // or any other constant int
  }
}
```

- this will cause a hashed container to put all **PhoneNumber**s in the same bucket

A Slightly Better `hashCode()`

```
public final class PhoneNumber {
  // attributes, constructors, methods ...

  @Override public int hashCode()
  {
    return (int)(this.getAreaCode() +
                 this.getExchangeCode() +
                 this.getStationCode());
  }
}
```

- the basic idea is generate a hash code using the attributes of the object
- it would be nice if two distinct objects had two distinct hash codes
  - but this is not required; two different objects can have the same hash code
- it is required that:
  1. if **`x.equals(y)`** then **`x.hashCode() == y.hashCode()`**
  2. **`x.hashCode()`** always returns the same value if **`x`** does not change its state

Something to Think About

- what do you need to be careful of when putting a mutable object into a **`HashSet`**?
  - can you avoid the problem by using immutable objects?

compareTo

Comparable Objects

- many value types have a natural ordering
  - that is, for two objects **`x`** and **`y`**, **`x`** is less than **`y`** is meaningful
    - **`Short`**, **`Integer`**, **`Float`**, **`Double`**, etc
    - **`String`**s can be compared in dictionary order
    - **`Date`**s can be compared in chronological order
    - you might compare **`Vector2D`**s by their length
    - **`Die`**s can be compared by their face value
- if your class has a natural ordering, consider implementing the **`Comparable`** interface
  - doing so allows clients to sort arrays or **`Collection`**s of your object

Interfaces

- an interface is (usually) a group of related methods with empty bodies
  - the `Comparable` interface has just one method

```
public interface Comparable<T>
{
  int compareTo(T t);
}
```

- a class that implements an interfaces promises to provide an implementation for every method in the interface

`compareTo()`

- Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Throws a **`ClassCastException`** if the specified object type cannot be compared to this object.

Die compareTo()

```
public class Die implements Comparable<Die> {
  // attributes, constructors, methods ...


  public int compareTo(Die other) {
    int result = 0;
    if (this.getValue() < other.getValue()) {
      result = -1;
    }
    else if (this.getValue() > other.getValue()) {
      result = 1;
    }
    return result;
  }
}
```

Die compareTo()

- the following also works for the Die class, but is dangerous in general:

```
public int compareTo(Die other) {
  int result = this.getValue() - other.getValue();
  return result;
}
```

Comparable Contract

1. the sign of the returned **`int`** must flip if the order of the two compared objects flip
   - if `x.compareTo(y) > 0` then `y.compareTo(x) < 0`
   - if `x.compareTo(y) < 0` then `y.compareTo(x) > 0`
   - if `x.compareTo(y) == 0` then `y.compareTo(x) == 0`

Comparable Contract

2. **`compareTo()`** must be transitive
   - if `x.compareTo(y) > 0 && y.compareTo(z) > 0` then `x.compareTo(z) > 0`
   - if `x.compareTo(y) < 0 && y.compareTo(z) < 0` then `x.compareTo(z) < 0`
   - if `x.compareTo(y) == 0 && y.compareTo(z) == 0` then `x.compareTo(z) == 0`

Comparable Contract

3. if **`x.compareTo(y) == 0`** then the signs of **`x.compareTo(z)`** and **`y.compareTo(z)`** must be the same

Consistency with equals

- an implementation of **`compareTo()`** is said to be consistent with **`equals()`** when

  if **`x.compareTo(y) == 0`** then

  `x.equals(y) == true`

  - and

  if **`x.equals(y) == true`** then

  `x.compareTo(y) == 0`

Not in the Comparable Contract

- it is not required that **`compareTo()`** be consistent with **`equals()`**
  - that is

    if **`x.compareTo(y) == 0`** then

    `x.equals(y) == false` is acceptable
  - similarly

    if **`x.equals(y) == true`** then

    `x.compareTo(y) != 0` is acceptable

- try to come up with examples for both cases above

Implementing `compareTo`

- implementing **`compareTo`** is similar to implementing equals
- you need to compare all of the fields
  - starting with the field that is most significant for ordering purposes and working your way down

PhoneNumber compareTo()

```
public class PhoneNumber implements Comparable<PhoneNumber> {
  // attributes, constructors, methods ...


  public int compareTo(PhoneNumber other) {
    int result = 0;
    result = this.getAreaCode() - other.getAreaCode();
    if (result == 0) {
      result = this.getExchangeCode() - other.getExchangeCode();
    }
    if (result == 0) {
      result = this.getStationCode() - other.getStationCode();
    }
    return result;
  }
}
```

Implementing **compareTo**

- if you are comparing fields of type **float** or **double** you should use **Float.compare** or **Double.compare** instead of **<**, **>**, or **==**

- if your **compareTo** implementation is broken, then any classes or methods that rely on **compareTo** will behave erratically
  - **TreeSet**, **TreeMap**
  - many methods in the utility classes **Collections** and **Arrays**

Mixing Static and Non-Static

`static` Fields

- a field that is **`static`** is a per-class member
  - only one copy of the field, and the field is associated with the class
    - every object created from a class declaring a static field shares the same copy of the field
- static fields are used when you really want only one common instance of the field for the class
  - less common than non-static fields

Example

- a textbook example of a static field is a counter that counts the number of created instances of your class

```
// adapted from Sun's Java Tutorial
public class Bicycle {
  // some other fields here...
  private static int numberOfBicycles = 0;

  public Bicycle() {
    // set some attributes here...
    Bicycle.numberOfBicycles++;
  }

  public static int getNumberOfBicyclesCreated() {
    return Bicycle.numberOfBicycles;
  }
}
```

note:
not this.numberOfBicycles++

- another common example is to count the number of times a method has been called

```
public class X {

  private static int numTimesXCalled = 0;
  private static int numTimesYCalled = 0;

  public void xMethod() {
    // do something... and then update counter
    ++X.numTimesXCalled;
  }

  public void yMethod() {
    // do something... and then update counter
    ++X.numTimesYCalled;
  }
}
```

Mixing Static and Non-static Fields

- a class can declare static (per class) and non-static (per instance) fields
- a common textbook example is giving each instance a unique serial number
  - the serial number belongs to the instance
    - therefore it must be a non-static field

```
public class Bicycle {
  // some attributes here...
  private static int numberOfBicycles = 0;

  private int serialNumber;

  // ...
```

- how do you assign each instance a unique serial number?
  - the instance cannot give itself a unique serial number because it would need to know all the currently used serial numbers
- could require that the client provide a serial number using the constructor
  - instance has no guarantee that the client has provided a valid (unique) serial number

- the class can provide unique serial numbers using static fields
  - e.g. using the number of instances created as a serial number

```
public class Bicycle {
  // some attributes here...

  private static int numberOfBicycles = 0;
  private int serialNumber;

  public Bicycle() {
    // set some attributes here...
    this.serialNumber = Bicycle.numberOfBicycles;
    Bicycle.numberOfBicycles++;
  }
}
```

- a more sophisticated implementation might use an object to generate serial numbers

```
public class Bicycle {

  // some attributes here...
  private static int numberOfBicycles = 0;

  private static final
    SerialGenerator serialSource = new SerialGenerator();

  private int serialNumber;

  public Bicycle() {
    // set some attributes here...
    this.serialNumber = Bicycle.serialSource.getNext();
    Bicycle.numberOfBicycles++;
  }
}
```

Static Methods

- recall that a **`static`** method is a per-class method
  - client does not need an object to invoke the method
  - client uses the class name to access the method
- a **`static`** method can only use **`static`** fields of the class
  - **`static`** methods have no **`this`** parameter because a **`static`** method can be invoked without an object
  - without a **`this`** parameter, there is no way to access non-static fields
- non-static methods can use all of the fields of a class (including **`static`** ones)

```
public class Bicycle {
  // some attributes, constructors, methods here...

  public static int getNumberCreated()
  {
    return Bicycle.numberOfBicycles;
  }

  public int getSerialNumber()
  {
    return this.serialNumber;
  }

  public void setNewSerialNumber()
  {
    this.serialNumber = Bicycle.serialSource.getNext();
  }
}
```

static method
can only use
static attributes

non-static method
can use
non-static attributes

and static attributes

Mixing Static and Non-static

Singleton

Singleton Pattern

- “There can be only one.”

- Connor MacLeod, Highlander

Singleton Pattern

- a singleton is a class that is instantiated exactly once
- singleton is a well-known design pattern that can be used when you need to:
  1. ensure that there is one, and only one*, instance of a class, and
  2. provide a global point of access to the instance
     - any client that imports the package containing the singleton class can access the instance

[notes 3.4]

*or possibly zero

One and Only One

- how do you enforce this?
  - need to prevent clients from creating instances of the singleton class
    - **`private`** constructors
  - the singleton class should create the one instance of itself
    - note that the singleton class is allowed to call its own **`private`** constructors
    - need a **`static`** attribute to hold the instance

A Silly Example: Version 1

uses a public field that all clients can access

```
package xmas;

public class Santa
{
  // whatever fields you want for santa...

  public static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize attributes here... }

}
```

```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.INSTANCE.givePresent();
}
```

A Silly Example: Version 2

uses a private field; how do clients access the field?

```
package xmas;

public class Santa
{
  // whatever fields you want for santa...

  private static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize attributes here... }

}
```

Global Access

- how do clients access the singleton instance?
  - by using a static method
- note that clients only need to import the package containing the singleton class to get access to the singleton instance
  - any client method can use the singleton instance without mentioning the singleton in the parameter list

A Silly Example (cont)

```
package xmas;

public class Santa {
  private int numPresents;
  private static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize fields here... }

  public static Santa getInstance()
  { return Santa.INSTANCE; }

  public Present givePresent() {
    Present p = new Present();
    this.numPresents--;
    return p;
  }
}
```

uses a private field; how do clients access the field?

clients use a public static factory method

```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.getInstance().givePresent();
}
```

Enumerations

- an enumeration is a special data type that enables for a variable to be a set of predefined constants
- the variable must be equal to one of the values that have been predefined for it
  - e.g., compass directions
    - NORTH, SOUTH, EAST, and WEST
  - days of the week
    - MONDAY, TUESDAY, WEDNESDAY, etc.
  - playing card suits
    - CLUBS, DIAMONDS, HEARTS, SPADES
- useful when you have a fixed set of constants

A Silly Example: Version 3

```
package xmas;

public enum Santa
{
  // whatever fields you want for santa...

  INSTANCE;

  private Santa()
  { // initialize attributes here... }

}
```

singleton as an enumeration

will call the private default constructor same usage as public field (Version 1)

```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.INSTANCE.givePresent();
}
```

Singleton as an enumeration

- considered the preferred approach for implementing a singleton
  - for reasons beyond the scope of CSE1030
- all enumerations are subclasses of **`java.lang.Enum`**

Applications

- singletons should be uncommon
- typically used to represent a system component that is intrinsically unique
  - window manager
  - file system
  - logging system

Logging

- when developing a software program it is often useful to log information about the runtime state of your program
  - similar to flight data recorder in an airplane
  - a good log can help you find out what went wrong in your program
- problem: your program may have many classes, each of which needs to know where the single logging object is
  - global point of access to a single object == singleton
- Java logging API is more sophisticated than this
  - but it still uses a singleton to manage logging
  - java.util.logging

Lazy Instantiation

- notice that the previous singleton implementation always creates the singleton instance whenever the class is loaded
  - if no client uses the instance then it was created needlessly
- it is possible to delay creation of the singleton instance until it is needed by using lazy instantiation
  - only works for version 2

Lazy Instantiation as per Notes

```
public class Santa {
  private static Santa INSTANCE = null;

  private Santa()
  { // ... }

  public static Santa getInstance()
  {
    if (Santa.INSTANCE == null) {
      Santa.INSTANCE = new Santa();
    }
    return Santa.INSTANCE;
  }
}
```

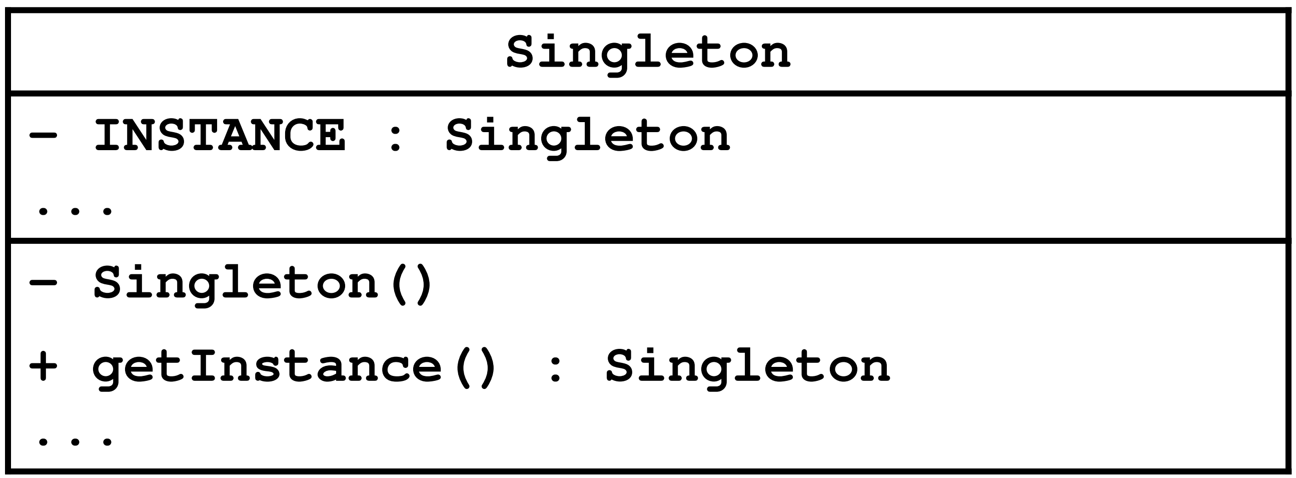
Mixing Static and Non-static

Multiton

Goals for Today

- Multiton
- review maps
- static factory methods

Singleton UML Class Diagram



One Instance per State

- the Java language specification guarantees that identical **`String`** literals are not duplicated

```
// client code somewhere

String s1 = "xyz";
String s2 = "xyz";

// how many String instances are there?
System.out.println("same object? " + (s1 == s2) );
```

  - prints: **`same object? true`**
- the compiler ensures that identical **`String`** literals all refer to the same object
  - a single instance per unique state

[notes 3.5]

Multiton

- a *singleton* class manages a single instance of the class
- a *multiton* class manages multiple instances of the class

- what do you need to manage multiple instances?
  - a collection of some sort

- how does the client request an instance with a particular state?
  - it needs to pass the desired state as arguments to a method

Singleton vs Multiton UML Diagram

Singleton
- INSTANCE : Singleton ...
- Singleton() + getInstance() : Singleton ...

Multiton
- instances : Map ...
- Multiton() + getInstance(Object) : Multiton ...

Singleton vs Multiton

- Singleton
  - one instance

```
private static final Santa INSTANCE = new Santa();
```

  - zero-parameter accessor

```
public static Santa getInstance()
```

Singleton vs Multiton

- Multiton
  - multiple instances (each with unique state)

```
private static final Map<String, PhoneNumber>
  instances = new TreeMap<String, PhoneNumber>();
```

  - accessor needs to provide state information

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
```

Map

- a map stores key-value pairs

```
Map<String, PhoneNumber>
```

key type — value type

- values are put into the map using the key

```
// client code somewhere
Map<String, PhoneNumber> m =
                      new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648"

m.put(key, ago);
```

[AJ 16.2]

- values can be retrieved from the map using only the key
  - if the key is not in the map the value returned is `null`

```
// client code somewhere
Map<String, PhoneNumber> m =
                      new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648";

m.put(key, ago);

PhoneNumber gallery = m.get(key);          // == ago
PhoneNumber art = m.get("4169796648");     // == ago

PhoneNumber pizza = m.get("4169671111");   // == null
```

- a map is not allowed to hold duplicate keys
  - if you re-use a key to insert a new object, the existing object corresponding to the key is removed and the new object inserted

```
// client code somewhere
Map<String, PhoneNumber> m = new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648";

m.put(key, ago);                                  // add ago
System.out.println(m);

m.put(key, new PhoneNumber(905, 760, 1911));    // replaces ago
System.out.println(m);
```

prints

```
{4169796648=(416) 979-6648}
{4169796648=(905) 760-1911}
```

Mutable Keys

- from
  http://docs.oracle.com/javase/7/docs/api/java/util/Map.html
  - Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map.

```java
public class MutableKey
{
  public static void main(String[] args)
  {
    Map<Date, String> m = new TreeMap<Date, String>();
    Date d1 = new Date(100, 0, 1);
    Date d2 = new Date(100, 0, 2);
    Date d3 = new Date(100, 0, 3);
    m.put(d1, "Jan 1, 2000");
    m.put(d2, "Jan 2, 2000");
    m.put(d3, "Jan 3, 2000");
    d2.setYear(101);                  // mutator
    System.out.println("d1 " + m.get(d1));  // d1 Jan 1, 2000
    System.out.println("d2 " + m.get(d2));  // d2 Jan 2, 2000
    System.out.println("d3 " + m.get(d3));  // d3 null
  }
}
```

don't mutate keys;
bad things will happen

change TreeMap to HashMap and see what happens

Making **PhoneNumber** a Multiton

1. multiple instances (each with unique state)

```
private static final Map<String, PhoneNumber>
    instances = new TreeMap<String, PhoneNumber>();
```

2. accessor needs to provide state information

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
```

- **`getInstance()`** will get an instance from **`instances`** if the instance is in the map; otherwise, it will create the new instance and put it in the map

Making `PhoneNumber` a Multiton

3. require private constructors
   - to prevent clients from creating instances on their own
     - clients should use **`getInstance()`**

4. require immutability of **`PhoneNumber`**s
   - to prevent clients from modifying state, thus making the keys inconsistent with the **`PhoneNumber`**s stored in the map
   - recall the recipe for immutability...

```java
public class PhoneNumber implements Comparable<PhoneNumber>
{
  private static final Map<String, PhoneNumber> instances =
                              new TreeMap<String, PhoneNumber>();

  private final short areaCode;
  private final short exchangeCode;
  private final short stationCode;

  private PhoneNumber(int areaCode,
                      int exchangeCode,
                      int stationCode)
  { // identical to previous versions }
```

why is validation not needed?

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
{
  String key = "" + areaCode + exchangeCode + stationCode;
  PhoneNumber n = PhoneNumber.instances.get(key);
  if (n == null)
  {
    n = new PhoneNumber(areaCode, exchangeCode, stationCode);
    PhoneNumber.instances.put(key, n);
  }
  return n;
}
// remainder of PhoneNumber class ...
```

```java
public class PhoneNumberClient {

  public static void main(String[] args)
  {
    PhoneNumber x = PhoneNumber.getInstance(416, 736, 2100);
    PhoneNumber y = PhoneNumber.getInstance(416, 736, 2100);
    PhoneNumber z = PhoneNumber.getInstance(905, 867, 5309);

    System.out.println("x equals y: " + x.equals(y) +
                       " and x == y: " + (x == y));

    System.out.println("x equals z: " + x.equals(z) +
                       " and x == z: " + (x == z));
  }
}
```

```
x equals y: true and x == y: true
x equals z: false and x == z: false
```

Bonus Content

- notice that Singleton and Multiton use a static method to return an instance of a class
- a static method that returns an instance of a class is called a *static factory method*
  - factory because, as far as the client is concerned, the method creates an instance
    - similar to a constructor

Static Factory Methods

- many examples
  - `java.lang.Integer`

    `public static Integer valueOf(int i)`
    - Returns a **`Integer`** instance representing the specified **`int`** value.
  - `java.util.Arrays`

    `public static int[] copyOf(int[] original, int newLength)`
    - Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

Java API Static Factory Methods

- `java.lang.String`

  `public static String format(String format, Object... args)`

  - Returns a formatted string using the specified format string and arguments.

- `cse1030.math.Complex`

  `public static Complex fromPolar(double mag, double angle)`

  - Returns a reference to a new complex number given its polar form.

- you can give meaningful names to static factory methods (unlike constructors)

```
public class Person {
  private String name;
  private int age;
  private int weight;

  public Person(String name, int age, int weight) { // ... }

  public Person(String name, int age) { // ... }

  public Person(String name, int weight) { // ... }
  // ...
}
```

illegal overload: same signature

```java
public class Person {  // modified from PEx's
  // attributes ...

  public Person(String name, int age, int weight) { // ... }

  public static Person withAge(String name, int age) {
    return new Person(name, age, DEFAULT_WEIGHT);
  }

  public static Person withWeight(String name, int weight) {
    return new Person(name, DEFAULT_AGE, weight);
  }
}
```

A Singleton Puzzle: What is Printed?

```
public class Elvis {
  public static final Elvis INSTANCE = new Elvis();
  private final int beltSize;
  private static final int CURRENT_YEAR =
    Calendar.getInstance().get(Calendar.YEAR);

  private Elvis() { this.beltSize = CURRENT_YEAR - 1930; }

  public int getBeltSize() { return this.beltSize; }

  public static void main(String[] args) {
    System.out.println("Elvis has a belt size of " +
                       INSTANCE.getBeltSize());
  }
}
```

from Java Puzzlers by Joshua Bloch and Neal Gafter