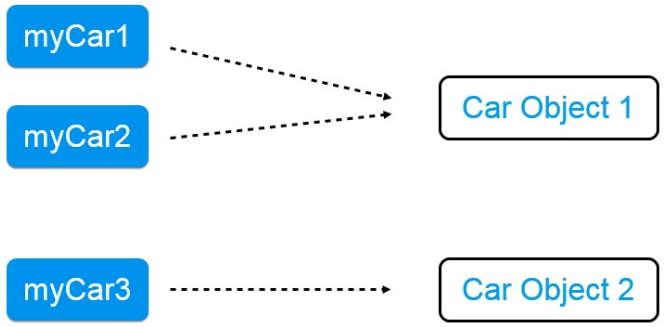
Identity and Equality

Identity vs. Equality

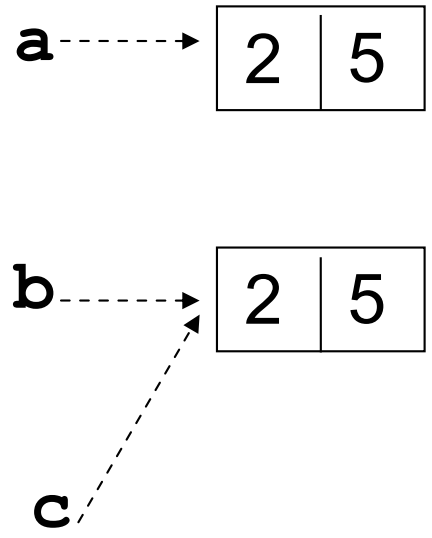


- Simple idea:
  - 2 objects are equal if they have the same value
  - 2 objects are equal if they are the same object
- Many subtleties
  - Same reference, or same value?
  - Same representation or same abstract value?
  - Equality in the presence of inheritance?
  - Does equality hold just now or is it eternal?
  - How can we implement equality efficiently?

Equality: `==` and `equals`

- Java uses the reference model for class types

```
class Point {
    int x; // x-coordinate
    int y; // y-coordinate
    Point(int x, int y) {
       this.x = x;
       this.y = y;
    }
}
a = new Point(2,5);
b = new Point(2,5);
c = b;
```



true or false? `a == b` ?

true or false? `b == c` ?

true or false? `a.equals(b)` ?

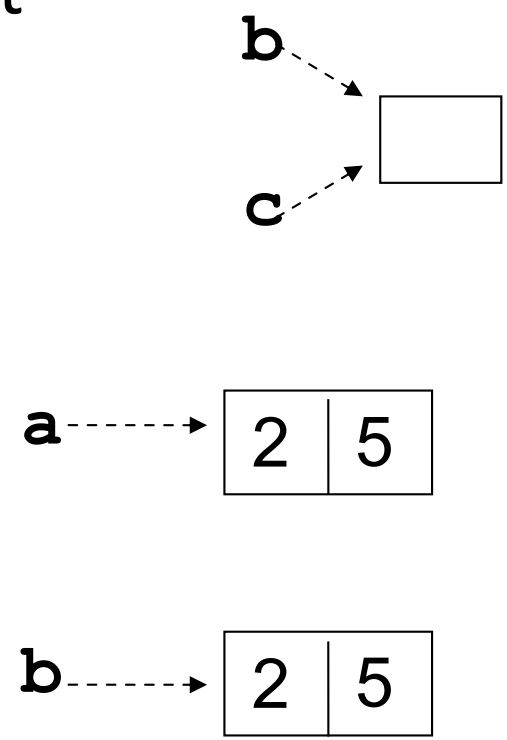
true or false? `b.equals(c)` ?

Assignment Operator "="

- = operator copies references for objects not “stuff”
- For primitives, it copies data

Equality: `==` and `equals`

- In Java, `==` tests for reference equality.
- Often value equality is what we want
- In our `Point` example, we want `a` to be "equal" to `b` because the `a` and `b` objects hold the same value
  - Need to override `Object.equals()`



Properties of Equality

- Equality is an equivalence relation
  - Reflexive `a.equals(a)`
  - Symmetric `a.equals(b)` ⇔ `b.equals(a)`
  - Transitive `a.equals(b)&& b.equals(c)` ⇨ `a.equals(c)`

- Is reference equality an equivalence relation?
  - Yes
  - Reflexive `a == a`
  - Symmetric `a == b` ⇔ `b == a`
  - Transitive `a == b && b == c` ⇨ `a == c`

`Object.equals` method

- `Object.equals` is very simple:
  - Point extends Object
  - all objects extend Object, implicitly
  - reference equality

```
public class Object {
      public boolean equals(Object obj) {
            return this == obj;
      }
}
```

PSoft spec for Object.equals

```
// requires none
// modifies none
// effects none
// throws none
// returns true if this == arg else false
// i.e., returns true if arg is same ref as this
```

Doesn't convey information about reflexivity, symmetry, or transitivity.

`Object.equals` Javadoc spec

Indicates whether some other object is "equal to" this one. The **`equals`** method implements an equivalence relation:

- It is *reflexive*: for any non-null reference value **`x`**, **`x.equals(x)`** should return true.
- It is *symmetric*: for any non-null reference values **`x`** and **`y`**, **`x.equals(y)`** should return true if and only if **`y.equals(x)`** returns true.
- It is *transitive*: for any non-null reference values **`x`**, **`y`**, and **`z`**, if **`x.equals(y)`** returns true and **`y.equals(z)`** returns true, then **`x.equals(z)`** should return true.
- It is *consistent*: for any non-null reference values **`x`** and **`y`**, multiple invocations of **`x.equals(y)`** consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

`Object.equals` Javadoc spec

For any non-null reference value **`x`**, **`x.equals(null)`** should return false.

The **`equals`** method for class Object implements the most discriminating possible (i.e., the strongest) *equivalence* relation on objects; that is, for any non-null reference values **`x`** and **`y`**, this method returns true if and only if **`x`** and **`y`** refer to the same object (**`x == y`** has the value true)…

Parameters:

`obj` - the reference object with which to compare.

Returns:

true if this object is the same as the **`obj`** argument;

false otherwise.

See Also:

`hashCode(), HashMap`

The **`Object.equals`** Spec

- Why this complex specification? Why not just
  - returns: true if obj == this, false otherwise
- Object is the superclass for all Java classes
  - The specification of **`Object.equals`** must be as weak (i.e., general) as possible
- Subclasses must be substitutable for Object
  - Thus, subclasses need to provide stronger **`equals`!**
  - No subclass can weaken **`equals`** and still be substitutable for Object!
  - Javadoc spec lists the properties of equality, the weakest possible specification of **`equals`**

Adding **equals**

```
public class Duration {
    private final int min;
    private final int sec;
    public Duration(int min, int sec) {
        this.min = min;
        this.sec = sec;
    }
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); // prints?
```

First Attempt to Add **`equals`**

```
public class Duration {
    public boolean equals(Duration d) {
      return
        this.min == d.min && this.sec == d.sec;
    }
}
Duration d1 = new Duration(10, 5);
Duration d2 = new Duration(10, 5);
System.out.println(d1.equals(d2));
```

Yields what?

- Is **`equals`** reflexive, symmetric and transitive?
- This **`equals`** is not quite correct. Why?

What About This?

```
public class Duration {
    public boolean equals(Duration d) {
       return
          this.min == d.min && this.sec == d.sec;
    }
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
```

Yields what?

`d1`'s compile-time type is `Object`.
`d1`'s runtime type is `Duration`.

Compiler looks at `d1`'s compile-time type.
Chooses signature `equals(Object)`.

What's wrong with Duration.equals()?

- It's an overload, not an override
- Overloading happens at compile time
- There are now 2 different versions of .equals()
  - One from Duration
  - One from Object
- Because d1 and d2 are of declared (compile-time) type Object, the call d1.equals(d2) resolves to equals(Object) at compile time.
- At runtime, it calls Object.equals(Object).
  - Probably not what the client expects

Java Overriding vs. Overloading

- Method overloading is when two or more methods in the same class have the exact same name but different parameters
  - When overloading, one changes either the type or the number of parameters for a method that belongs to the same class. Overriding means that a method inherited from a parent class will be changed.
  - Happens at compile time
- Method overriding is when a derived class requires a different definition for an inherited method,
  - The method can be redefined in the derived class.
  - *In overriding* a method, arguments remain exactly the same
    - The method name, the number and types of parameters all remain the same
  - the method definition – what the method does is changed slightly to fit in with the needs of the child class.
  - The return type must be the same as, or a subtype of the return type declared in the overridden method in the superclass.
  - Happens at runtime

Java Override

- Java supports covariant return types for overridden methods.
  - This means an overridden method may have a *more* specific return type.
  - As long as the new return type is assignable to the return type of the method you are overriding, it's allowed.
- A method declaration d1 with return type R1 is return-type-substitutable for another method d2 with return type R2, if and only if the following conditions hold:
  - If R1 is void then R2 is void.
  - If R1 is a primitive type, then R2 is identical to R1.
  - If R1 is a reference type then:
    - R1 is the same or a subtype of R2

Java Rules for Overrides

- The overriding method can not have a more restrictive access modifier than the method being overridden but it can be less restrictive.
  - Access modifier: private, public etc.
- The argument list must exactly match that of the overridden method. If it doesn't, you are overloading the method.
  - If the argument list is not the same, it's an overload.
- The return type must be the same as, or a subtype of the return type declared in the overridden method in the superclass.
- If superclass method does not declare any exception, then subclass overridden method cannot declare a checked exception, but it can declare unchecked exceptions.
- If superclass method throws an exception, then the subclass overridden method can throw the same or a subclass of the exception or no exception, but must not throw a parent exception of the exception thrown by superclass method.
  - That is, if the superclass method throws an object of IOException class, then subclass method can either throw the same exception, or can throw no exception, but it cannot throw an object of Exception class (parent of IOException class).
- A final or static method can not be overridden.

A More Correct **equals**

```
@Override
public boolean equals(Object o) {
  if (! (o instanceof Duration) )
     return false;
  Duration d = (Duration) o;
  return this.min == d.min && this.sec == d.sec;
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
```

Yields what?

Add a Nano-second Field

```
public class NanoDuration extends Duration {
  private final int nano;
  public NanoDuration(int min,
                      int sec,
                      int nano){
    super(min,sec);// initializes min&sec
    this.nano = nano;
  }
}
```

- What if we don't add **`NanoDuration.equals?`**

`(Assume Duration.equals as in previous slide)`

First Attempt at **`NanoDuration.equals`**

```
public boolean equals(Object o) {
  if (! (o instanceof NanoDuration) )
        return false;
  NanoDuration nd = (NanoDuration) o;
  return super.equals(nd) && nd.nano == nano;
}

Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(5,10);
d1.equals(d2);
d2.equals(d1);
```

Yields what?

Yields what?

Possible Fix for **`NanoDuration.equals`**

```
public boolean equals(Object o) {
  if (! (o instanceof Duration) )
      return false;
  if (! (o instanceof NanoDuration) )
     return super.equals(o);//compare without nano
                                  // Is this what we want?
  NanoDuration nd = (NanoDuration) o;
  return super.equals(o) && nd.nano == nano;
}
```

- Does it fix the symmetry bug?
- What can go wrong?

Possible Fix for **`NanoDuration.equals`**

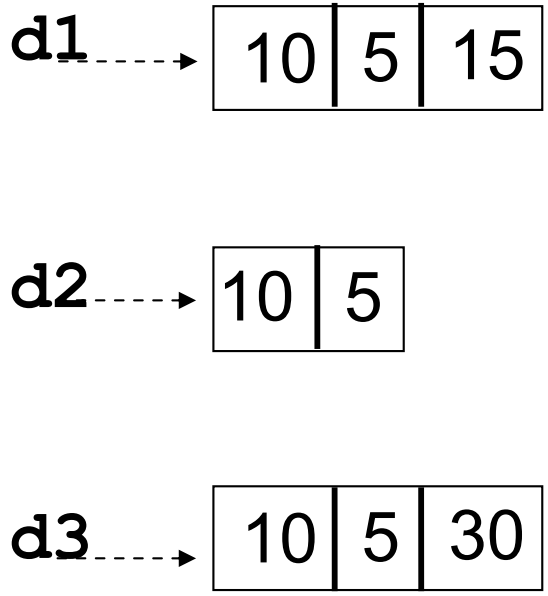
```
Duration d1 = new NanoDuration(10,5,15);
Duration d2 = new Duration(10,5);
Duration d3 = new NanoDuration(10,5,30);
```

`d1.equals(d2);` Yields what?

`d2.equals(d3);` Yields what?

`d1.equals(d3);` Yields what?

`equals` is not transitive!



One Solution: Checking Exact Class, Instead of **`instanceof`**

```
class Duration {
  public boolean equals(Object o) {
    if (o == null) return false;
    if ( !o.getClass().equals(getClass()) )
                    return false;
    Duration d = (Duration) o;
    return d.min == min && d.sec == sec;
  }
}
```

- Problem: every subclass must implement **`equals`**; sometimes, we want to compare distinct classes!

Another Solution: Composition

```
public class NanoDuration {
  private final Duration duration;
  private final int nano;
  …
}
```

Composition does solve the **`equals`** problem: **`Duration`** and **`NanoDuration`** are now unrelated, so we'll never compare a **`Duration`** to a **`NanoDuration`**

Problem: Can't use **`NanoDuration`** instead of **`Duration.`** Can't reuse code written for **`Duration.`**

A Reason to Avoid Subclassing Concrete Classes. More later

- In the JDK, subclassing of concrete classes is relatively rare. When it happens, there can be problems

- One example: **`Timestamp extends Date`**
  - Extends **`Date`** with a nanosecond value
  - But **`Timestamp`** spec lists several caveats
    - E.g., **`Timestamp.equals(Object)`** method is not symmetric with respect to **`Date.equals(Object)`**
      - (the symmetry problem we saw on the previous slides)

Abstract Classes

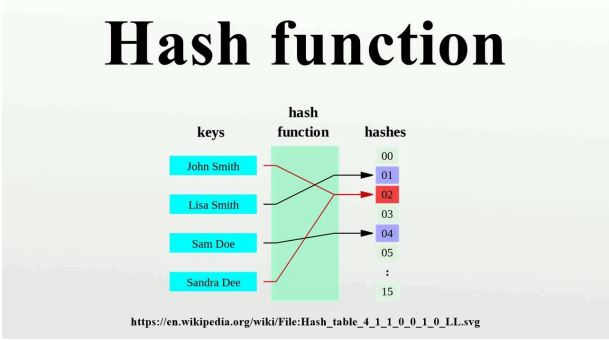
- Prefer subclassing abstract classes
  - "Superclasses" cannot be instantiated

- There is no equality problem if superclass cannot be instantiated!
  - E.g., if **`Duration`** were abstract, the issue of comparing **`Duration`** and **`NanoDuration`** never arises

Our Story So Far...

- Java supports two kinds of equality for objects
  - Reference equality
    - Do these 2 variables refer to the same object?
  - Value equality
    - Do these 2 variables refer to two objects with the same value?
    - What does "the same value" mean?
  - If you don't override .equals(), you get Object.equals()
    - Object.equals() is reference equality
- Equality is an equivalence relationship
- Equality can get complicated in the context of inheritance

Hash Function



- A function that maps an object to a number
- For data storage, the number is used to index into a fixed-size table called a *hash table*.
- Allows data storage and retrieval applications to access data in a small and nearly constant time per retrieval
  - Worst case can be bad

The `int hashCode` Method

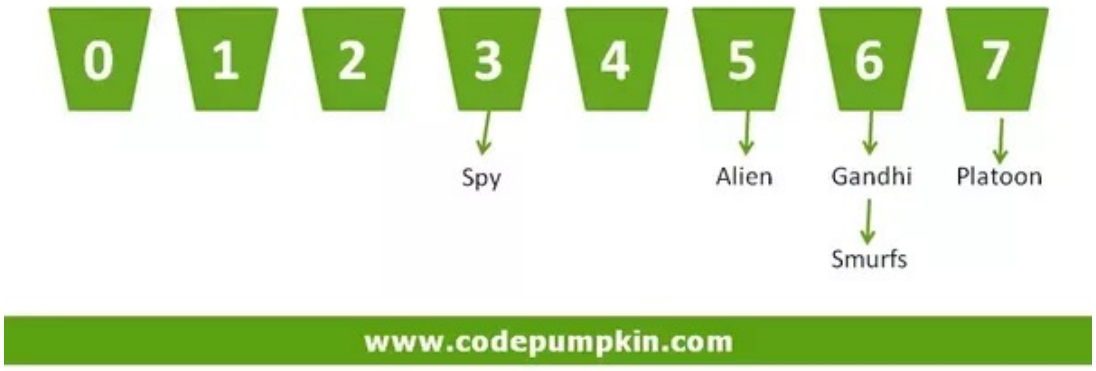
- **`hashCode`** computes an index for the object (to be used in hashtables)
- Javadoc for **`Object.hashCode()`**:
  - "Returns a hash code value of the object. This method is supported for the benefit of hashtables such as those provided by **`HashMap`**."
  - Self-consistent: **`o.hashCode() == o.hashCode()`**
    … as long as **`o`** does not change between the calls
  - Consistent with **`equals()`** method: **`a.equals(b) => a.hashCode() == b.hashCode()`**
  - If a.equals(b), a and b must have the same hashCode
    - Not necessarily true the other way around
  - Collections such as HashMap calculate *unicity* using .equals and .hashCode

Hash Code

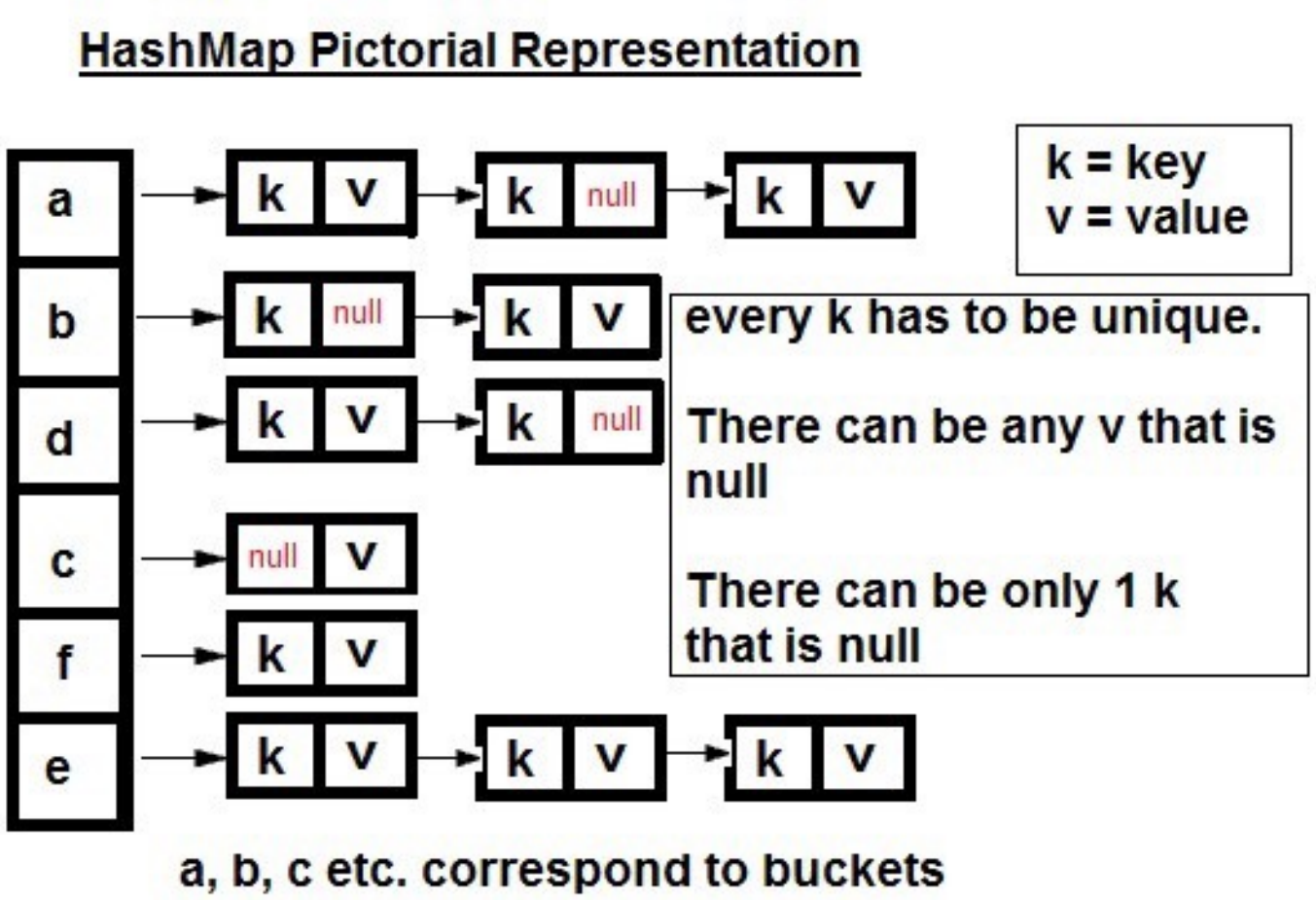
- hashCode() is used for *bucketing* in Hash implementations
  - HashMap, HashTable, HashSet, etc.
- The value received from hashCode() is used to determine the bucket for storing elements of the set/map.
  - The bucket is the address of the element inside the set/map.
- When you do contains() it will take the hash code of the element, then look for the bucket where hash code points.
    - Multiple objects can have the same hash code
    - It uses the equals() method to evaluate if the objects are equal
      - Only looks in the current bucket
    - Then decide if contains() is true or false

HashSet

Key	HashCode
Gandhi	6
Platoon	7
Alien	5
Smurfs	6
Spy	3



HashMap


HashMap Pictorial Representation
a
b
d
c
f
e
k
v
null
k = key
v = value
every k has to be unique.
There can be any v that is null
There can be only 1 k that is null
a, b, c etc. correspond to buckets

Hash Code

- By definition, if two objects are equal, their hash code must also be equal.
- If you override the equals() method, you change the way two objects are equated
  - Object's implementation of hashCode() is no longer valid.
  - Therefore, if you override the equals() method, you must also override the hashCode() method as well.
- We didn't do this for Duration to keep the description simple
  - But we should have

The **`Object.hashCode`** Method

- **`Object.hashCode`**'s implementation returns a distinct integer for each distinct object, typically by converting the object's address into an integer
- **`hashCode`** must be consistent with equality
  - **`equals`** and **`hashCode`** are used in hashtables
  - If **`hashCode`** is inconsistent with **`equals`**, the hashtable behaves incorrectly
  - Rule: if you override **`equals`**, override **`hashCode`**; must be consistent with **`equals`**
- **`Eclipse has an option to automatically generate a hashCode() method.`**

Rules for HashCodes

- During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value, provided no information used in **equals(Object)** comparison on the Object is modified.
  - It is not necessary that this Integer value remain the same from one execution of the application to another execution of the same application.
- If two Objects are equal, according to the **equals(Object)** method, then hashCode() method must produce the same Integer on each of the two Objects.
- If two Objects are unequal, according to the **equals(Object)** method, it is not necessary that the Integer value produced by the hashCode() method on each of the two Objects be distinct.
  - Objects which are distinct by the **equals(Object)** method may still compute the same hashCode() value
  - It can be same but producing a distinct Integer on each of the two Objects is better for improving the performance of hashing based Collections like HashMap, HashTable…etc.

Implementations of **`hashCode`**

Remember, we defined **`Duration.equals(Object)`**

```
public class Duration {
```

Choice 1: don't override, inherit **`hashCode`** from Object

Choice 2: `public int hashCode() { return 1; }`

Choice 3: `public int hashCode() { return min; }`

Choice 4: `public int hashCode() { return min+sec; }`

```
}
```

hashCode Must Be Consistent with **equals**

- Suppose we change **Duration.equals**

// Returns true if **o** and **this** represent the same number of

// seconds

```
public boolean equals(Object o) {
    if (!(o instanceof Duration)) return false;
    Duration d = (Duration) o;
    return 60*min+sec == 60*d.min+d.sec;
}
```

- Will **min+sec** for **hashCode** still work?
  - Problem: 0 min 90 secs and 1 min 30 secs are equal by this method, but hash1 = 90 and hash2 = 31. Equal objects have different hashCodes.

Equality, Mutation and Time

- If two objects are equal now, will they always be equal?
  - In mathematics, the answer is “yes”
    - Given that the object is not a function of time
  - In Java, the answer is “you choose”
  - The Object spec does not specify this
- For immutable objects
  - Abstract value never changes, equality is eternal
- For mutable objects
  - We can either compare abstract values now, or
  - be eternal (can’t have both since value can change)

`StringBuffer` Example

- StringBuffer is mutable, and takes the eternal approach

```
StringBuffer s1 = new StringBuffer("hello");
StringBuffer s2 = new StringBuffer("hello");
System.out.println(s1.equals(s1)); // true
System.out.println(s1.equals(s2)); // false
```

- **`equals`** is just reference equality (==). This is the only way to ensure eternal equality for mutable objects
  - Eternal means that as long as neither object changes, they stay equal or not-equal

Date Example

- **Date** is mutable, and takes the "compare values now" approach

```
Date d1 = new Date(0); //Jan 1, 1970 00:00:00 GMT
Date d2 = new Date(0);
System.out.println(d1.equals(d2)); // true
d2.setTime(10000); //some time later
System.out.println(d1.equals(d2)); // false
```

Behavioral and Observational Equivalence

- Two objects are "behaviorally equivalent" if there is no sequence of operations that can distinguish them
  - This is "eternal" equality
  - Two Strings with same content are behaviorally equivalent, two StringBuffers with same content are not
  - They cannot be distinguished by *any* observation
  - The two objects will "behave" the same, in this and all future states.

- Two objects are "observationally equivalent" if there is no sequence of observer operations that can distinguish them
  - We are excluding mutators
  - Excluding ==
  - Two Strings, Dates, or StringBuffers with same content are observationally equivalent.
  - Cannot be distinguished by observation *that doesn't change the state of the objects*, i.e., by calling only observer, producer, and creator methods.
  - They look the same

Equality and Mutation

- Date class implements observational equality
- We can violate the rep invariant of a Set container (rep invariant: there are no duplicates in set) by mutating after insertion

```
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1);
s.add(d1);
s.add(d2);
d2.setTime(0); // mutation after d2 already in the Set!
for (Date d : s) { // prints 2 identical dates
   System.out.println(d);
}
```

It's even worse

```
List<String> list = new ArrayList<String>();
list.add("cat");

Set<List<String>> set = new HashSet<List<String>>();
set.add(list);

System.out.println(set.contains(list));  // true

list.add("dog");
System.out.println(set.contains(list)); //false

for (List<String> l : set) {
    System.out.println(set.contains(l));  // false
}
```

What happened?

- List<String> is a mutable object.
- Mutations affect the result of equals() and hashCode().
- When the list is first put into the HashSet, it is stored in the hash bucket corresponding to its hashCode() result at that time.
- When the list is subsequently mutated, its hashCode() changes, but HashSet doesn’t realize it.
- set.contains() looks in the wrong bucket
  - So it can not be found.

Equality and Mutation

- Be very careful with elements of Sets
- Ideally, elements should be immutable objects, because immutable objects guarantee behavioral equivalence
- Java spec for Sets warns about using mutable objects as set elements
    - from the specification of java.util.Set:
        - Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.

- Same problem applies to keys in maps

Equality and Mutation

- Sets assume hash codes don't change

```
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1000); //  later
s.add(d1);
s.add(d2);
d2.setTime(10000);
s.contains(d2); // false
s.contains(new Date(10000));    // false
s.contains(new Date(1000));    // false again
```

What happened?

- Date is deprecated
  - Use Calendar instead.
- Set.contains() uses Date.equals().
  - Date overrides equals
- Why is s.contains(d2) false?
  - Because the hashCode has changed, but s doesn't know it.
  - s.contains(d2) is looking in the 10000
    - But d2 is still in the 1000 bucket
  - s.contains(new Date(1000)) fails because contains looks in the 1000 bucket but there's no longer a Date(1000) in 1000 bucket
    - We changed it
  - s.contains(new Date(10000)); fails because it looks in the 10000 bucket, but d2 is in the 1000 bucket

- These sorts of problems are hard to track down. This is why you should avoid sets of mutable objects whenever possible.

Equality and Mutation

- Redefining **`equals`** and **`hashCode`** makes most sense for immutable, "value", objects
  - E.g., String, RatNum
- Be careful with **`equals`** and **`hashCode`** on mutable objects
  - From spec of Object.equals: It is *consistent*: for any non-null reference values **`x`** and **`y`**, multiple invocations of **`x.equals(y)`** consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

Equality and Mutation

- From JavaDoc
  - Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.
    - See Russell's Paradox https://en.wikipedia.org/wiki/Russell's_paradox
- HashSet.contains() uses hashCode()
- HashSet.contains() method relies on hash values to stay immutable
  - There is an assumption that the hashCode() does not change
- contains() computes the hashCode() of the object it is looking for
  - It searches only the *bucket* that contains the hash value
- The moral
  - If you put mutable objects in a Set, don't modify them and expect operations like contains() to work as expected.

Mutation and hash codes

- Sets assume that the hash codes don't change
- Mutation can break this assumption

```
List<String> friends =
      new LinkedList<String>(Arrays.asList("yoda", "zaphod"));
List<String> enemies =
      new LinkedList<String>(Arrays.asList("Darth Vader", "Joker"));

Set<List<String>> h = new HashSet<List<String>>();
h.add(friends);
h.add(enemies);
friends.add("BatMan");

System.out.println(h.contains(friends));
System.out.println();
for (List<String> lst : h) {
      System.out.println(lst.equals(friends));
}  // one "true" will be printed - inconsistent
```

Equality and Sets

```
Set<String> set1 = new HashSet <String>();
Set<String> set2 = new TreeSet <String>();
for (String s : "hi how are you".split(" ")) {
     set1.add(s);
     set2.add(s);}
System.out.println(set1.equals(set2));  // true
```

- Objects of different types are usually not equals()

to each other.

- But the documentation for the Set interface specifies:

  Returns true if the specified object is also a set,
  the two sets have the same size,
  and every member of the specified set is contained in this set
  (or equivalently, every member of this set is contained in the specified set).

This definition ensures that the equals method works properly across different implementations of the set interface.

Implementing **`equals`** Efficiently

- **`equals`** can be expensive!

- How can we speed-up **`equals`**?

```
public boolean equals(Object o) {
   if (this == o) return true;
  // class-specific prefiltering (e.g.,
  // compare file size if working with files)
  // Lastly, compare fields (can be expensive)
}
```

Example: A Naïve **`RatPoly.equals`**

```
public boolean equals(Object o) {
  if (o instanceof RatPoly) {
    RatPoly rp = (RatPoly) o;
    int i=0;
    while (i<Math.min(rp.c.length,c.length)) {
      if (rp.c[i] != c[i]) // Assume int arrays
        return false;
      i = i+1;
    }
    if (i != rp.c.length || i != c.length) return false;
    return true;
  else
    return false;
}
```

Example: Better **`equals`**

```
public boolean equals(Object o) {
  if (o instanceof RatPoly) {
    RatPoly rp = (RatPoly) o;
    if (rp.c.length != c.length)
      return false; // prefiltering
    for (int i=0; i < c.length; i++) {
      if (rp.c[i] != c[i])
        return false;
    }
    return true;
  }
  else
    return false;
}
```

Implementing **hashCode**

// returns: the **hashCode** value of this String

```
public int hashCode() {
    int h = this.hash; // rep. field hash
    if (h == 0) {      // caches the hashcode
       char[] val = value;
       int len = count;
       for (int i = 0; i < len; i++) {
           h = 31*h + val[i];
       }
       this.hash = h;
     }
    return h;
}
```

This works only for immutable objects!

Rep Invariant, AF and Equality

- With ADTs we compare abstract values, not rep
- Usually, many valid reps map to the same abstract value
  - If Concrete Object (rep) and Concrete Object' (rep') map to the same Abstract Value, then Concrete Object and Concrete Object' must be **`equal`**
- A stronger rep invariant shrinks the domain of the AF and simplifies **`equals`**

Example: Line Segment

```
class LineSegment {
// Rep invariant:
// !(x1=x2 && y1=y2)
   float x1,y1;
   float x2,y2;
…
}
// equals must
// return true for
// {x1:1,y1:2,x2:4,y2:5}
// and {4,5,1,2}
```

```
class LineSegment {
// Rep invariant:
// x1<x2 ||
// x1=x2 && y1<y2
   float x1,y1;
   float x2,y2;
…
}
// equals is simpler:
// {4,5,1,2} is not
// valid rep anymore
```

Rules for overriding equals()

- Overriding equality seems easy but many ways to get it wrong
- Obey the general contract
- Don’t do it if
  - Each instance of the class is inherently unique.
  - You don’t care whether the class provides a “logical equality” test
    - Random numbers
  - A superclass has already overridden equals, and the behavior inherited from the superclass is adequate for this class.
    - Set inherits its equals from AbstractSet
  - The class is private or package-private, and you are certain that its equals method will never be invoked.

Rules for overriding equals()

- If you need to:
  - Use the == operator to check if the argument is a reference to this object
  - Use the instanceof operator to check if the argument is of the correct type
  - Cast the argument to the correct type
  - For each “significant” field in the class, check to see if that field of the argument matches the corresponding field of this object.
  - When you are finished writing your equals method, ask yourself four questions: Is it reflexive, is it symmetric, is it transitive, and is it consistent?
  - Always override hashCode when you override equals
  - Don’t substitute another type for Object in the equals declaration.
  - Eclipse can generate Java hashCode and equals methods
    - Source->Generate hashCode() and equals()‘.