CSE 331

Hash codes; annotations

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Surprising result #1

```
Point p = new Point(3, 4);
Set<Point> set = new HashSet<Point>();
set.add(p);
System.out.println(set.contains(new Point(3, 4))); // true

p.translate(2, 2);
System.out.println(set.contains(new Point(5, 6))); // false
```

- Where did `p` go? What is wrong?

Hashing

- **hash**: To map a value to a specific integer index.
  - **hash table**: An array that stores elements via hashing.
    - The internal data structure used by `HashSet` and `HashMap`.
  - **hash function**: An algorithm that maps values to indexes.
    - A possible hash function for integers: HF(i) → i % length

```
set.add(11);        // 11 % 10 == 1
set.add(49);        // 49 % 10 == 9
set.add(24);        // 24 % 10 == 4
set.add(7);         //  7 % 10 == 7
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

  - What is a hash function for a string? for other kinds of objects?

Efficiency of hashing

```
public static int HF(int i) {     // int hash function
    return Math.abs(i) % elements.length;
}
```

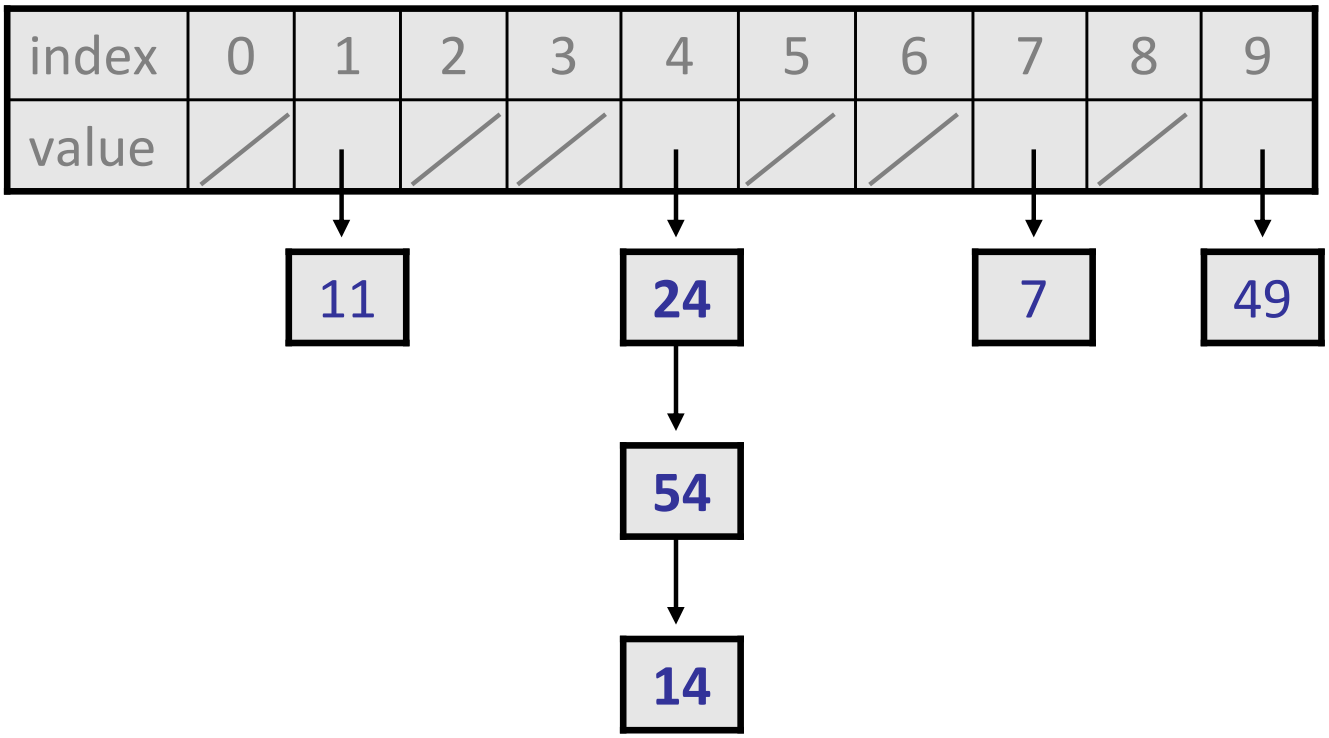
- Add: simply set `elements[HF(i)] = i;`
- Search: check if `elements[HF(i)] == i`
- Remove: set `elements[HF(i)] = 0;`

- Runtime of `add`, `contains`, and `remove`: **O(1)!**

- Are there any potential problems with hashing?
  - **collisions**: Multiple element values can map to the same bucket.

Chaining

- **chaining**: Resolving collisions by storing a list at each index.
  - Add/search/remove must traverse lists, but the lists are short.
  - Impossible to "run out" of indexes, unlike with probing.
  - Alternative to chaining: *probing* (choosing the next available index).



The hashCode method

- From the `Object` class:

  `public int` **`hashCode`**`()`

  Returns an integer hash code for this object.

  - We can call `hashCode` on *any object* to find the index where it "prefers" to be placed in a hash table.

Hash function for objects

```
public static int HF(Object o) {
    return Math.abs(o.hashCode()) % elements.length;
}
```

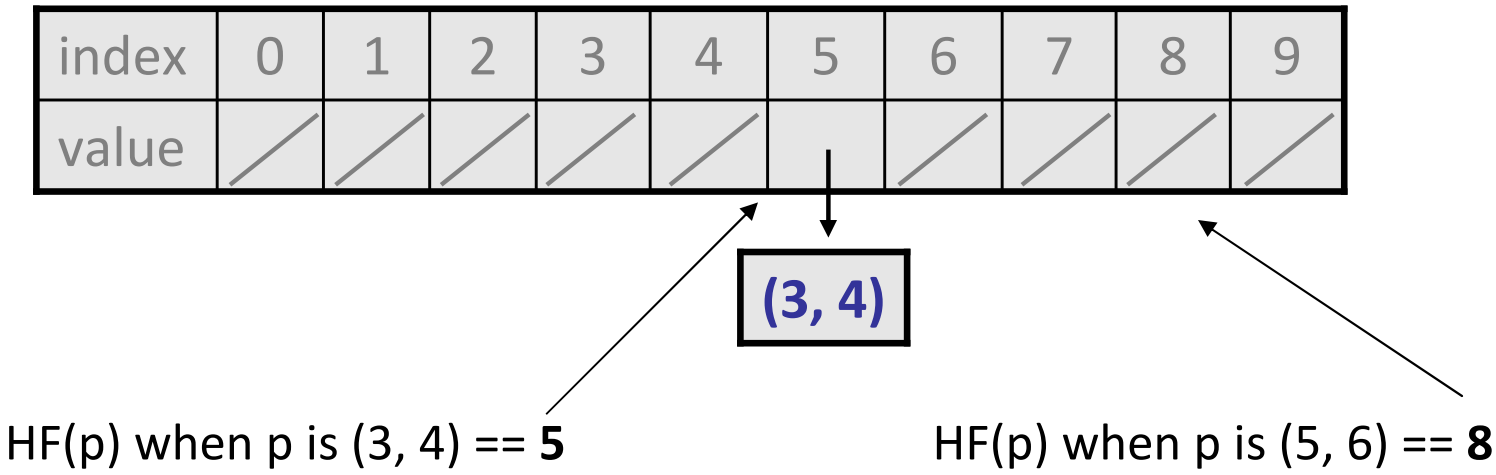
- Add: simply set `elements[HF(o)] = o;`
- Search: check if `elements[HF(o)].equals(o)`
- Remove: set `elements[HF(o)] = null;`

Surprising result #1

```
Point p = new Point(3, 4);
Set<Point> set = new HashSet<Point>();
set.add(p);
System.out.println(set.contains(new Point(3, 4))); // true

p.translate(2, 2);
System.out.println(set.contains(new Point(5, 6))); // false
```

- The code breaks because the point is put into a certain bucket when its state is (3, 4), but it isn't in the bucket that is returned when `hashCode` is called on an object with state (5, 6).



Surprising result #2

```
// assuming that Time is a class we have written,
// and that Time does have a proper equals method
Time t1 = new Time(11, 30, true);
Time t2 = new Time(11, 30, true);
Set<Time> set = new HashSet<Time>();
set.add(t1);
System.out.println(set.contains(t1));  // true
System.out.println(set.contains(t2));  // false
```

- What is wrong?

Implementing hashCode

- `hashCode`'s implementation depends on the object's type/state.
  - A String's `hashCode` method adds the ASCII values of its letters.
  - A Point's `hashCode` produces a weighted sum of its x/y coordinates.
  - A Double's `hashCode` converts the number into bits and returns that.
  - A collection's `hashCode` combines the hash codes of its elements.

- You can override `hashCode` in your classes.
  - ***Effective Java* Tip #9**:
    Always override `hashCode` when you override `equals`.

- The default implementation from class `Object` just uses the object's memory address to produce the integer code.
  - Why might this not be ideal?

The hashCode contract

- The general contract of `hashCode` is that it must be:
  - *Self-consistent* (produces the same results on each call):

    `o.hashCode() == o.hashCode()`

    ...so long as **`o`** doesn't change between the calls

  - *Consistent with equality*:

    `a.equals(b)` implies that

    `a.hashCode() == b.hashCode()`

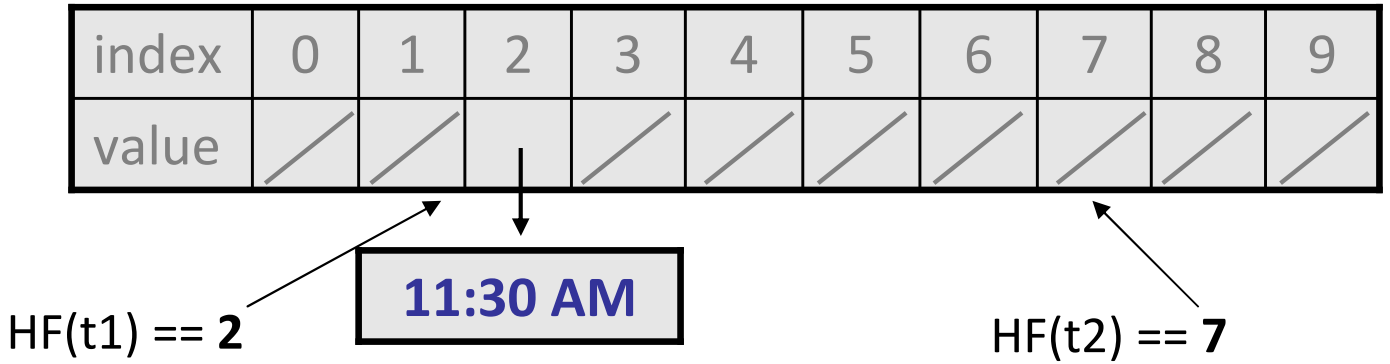
    `!a.equals(b)` does NOT necessarily imply that

    `a.hashCode() != b.hashCode()` *(why not?)*

Surprising result #2

```
// assuming that Time is a class we have written,
// and that Time does have a proper equals method
Time t1 = new Time(11, 30, "AM");
Time t2 = new Time(11, 30, "AM");
Set<Time> set = new HashSet<Time>();
set.add(t1);
System.out.println(set.contains(t1));  // true
System.out.println(set.contains(t2));  // false
```

- The code breaks because `Time` has no `hashCode` method, so each object's hash code is just its memory address. This is inconsistent with `equals` and so it cannot find seemingly equal objects in the set.



hashCode implementation 1

- Possible implementation of `hashCode` for `Time` objects:

```
public int hashCode() {
    return new Random().nextInt();
}
```

  - Does this meet the general contract of `hashCode`?
  - Is this a good `hashCode` function? Why or why not?
    - In what cases does this `hashCode` produce poor results?

hashCode implementation 2

- Possible implementation of `hashCode` for `Time` objects:

```
public int hashCode() {
    return 42;
}
```

  - Does this meet the general contract of `hashCode`?
  - Is this a good `hashCode` function? Why or why not?
    - In what cases does this `hashCode` produce poor results?

hashCode implementation 3

- Possible implementation of `hashCode` for `Time` objects:

```
public int hashCode() {
    return hour + minute;
}
```

  - Does this meet the general contract of `hashCode`?
  - Is this a good `hashCode` function? Why or why not?
    - In what cases does this `hashCode` produce poor results?

hashCode implementation 4

- Recommended implementation of `hashCode` for `Time` objects:

```
public int hashCode() {
    return 65531 * amPm.hashCode()
           + 67 * hour + minute;
}
```

  - All fields of the object should be incorporated into the hash code.
  - The code should weight each field by multiplying them by various prime numbers to reduce collisions between unequal objects.
    - e.g. Don't want 11:05 AM to collide with 5:11 PM if possible.
    - We prefer to multiply by primes because they wrap more unevenly when they exceed the array's size.

hashCode tricks

- If one of your object's fields is an object, call its `hashCode`:

```
public int hashCode() {              // TimeSpan
    return 65531 * amPm.hashCode() + ...;
}
```

- To incorporate an array, use `Arrays.hashCode`.

```
private String[] addresses;

public int hashCode() {
    return 3137 * Arrays.hashCode(addresses) + ...
}   // also Arrays.deepHashCode for multi-dim arrays
```

hashCode tricks 2

- To incorporate a `double` or `boolean`, use the `hashCode` method from the `Double` or `Boolean` wrapper classes:

```
public int hashCode() {              // BankAccount
    return 37 * new Double(balance).hashCode() +
     new Boolean(isCheckingAccount).hashCode() + ...;
}
```

- If your hash code is expensive to compute, consider caching it.

```
private int myHashCode = ...;    // pre-compute once

public int hashCode() {
    return myHashCode;
}
```

String's hashCode

- The `hashCode` function for `String` objects looks like this:

```
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

  - Early versions of the Java examined only the first 16 characters. For some common data this led to poor hash table performance.
  - As with any general hashing function, collisions are possible.
    - Example: "Ea" and "FB" have the same hash value.

Annotations

Annotations

- **annotation**: Markup that provides information to the compiler.
  - Can also be used for deployment-time or run-time processing.

- Common uses for annotations:
  - To detect problems or errors in code
  - To suppress compiler warnings
  - For unit tests, e.g. JUnit

Annotation usage

@AnnotationName

@AnnotationName(param=value, ..., param=value)

- Examples:

```
@SuppressWarnings
@Test(timeout=2000)
```

- An annotation can be placed on:
  - a class
  - a method
  - a field
  - a local variable, ...

Common annotations

- The following annotation types come with the JDK:

name	description
`@SuppressWarnings`	turn off compiler warnings
`@Override`	a superclass's method that is being overridden
`@Deprecated`	code that is discouraged from use
`@Documented`	sets an annotation to appear in Javadoc
`@Retention`	makes annotation data available at runtime

Using @Override

- Whenever you override a superclass's method, such as `equals` or `hashCode`, you should annotate it with `@Override`.
  - If you do, the compiler will produce an error if you don't override it properly (misspell the name, wrong parameter/return types, etc.)

```
@Override
public boolean equal(Object other) { ...
    // error; should be 'equals'
```

  - This also applies to methods you implement from an interface.

```
@Override
public int compareTo(Time other) { ...
```

Creating an annotation type

```
public @interface Name {}
```

Example:

```
public @interface GradingScript {}
...

@GradingScript
public class TestElection {...}
```

- Most programmers don't commonly need to create annotations.

An annotation with params

```
public @interface Name {
    type name();                    // parameters
    type name() default value;      // optional
}
```

Example:

```
public @interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String[] reviewers();
}
```

Using custom annotation

```
@ClassPreamble(
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    reviewers = {"Alice", "Bob", "Cindy"}
)
public class FamilyTree {
    ...
}
```

Prof. Ernst's annotations

- UW's own Prof. Michael Ernst and his research team have contributed a set of custom annotations that can be used to provide sophisticated type checking and nullness checking for Java:

name	description
`@Immutable`	a class of objects that cannot mutate
`@Readonly`	a temporarily unmodifiable value
`@Nullable`, `@NonNull`	a value for which null can/cannot be passed
`@Encrypted`, `@Untainted`	for security and encryption
`@GuardedBy`	for concurrency
`@Localized`	for internationalization
`@Regex`	for tagging regular expressions
`@Interned`	for flyweighted objects