Non-static classes

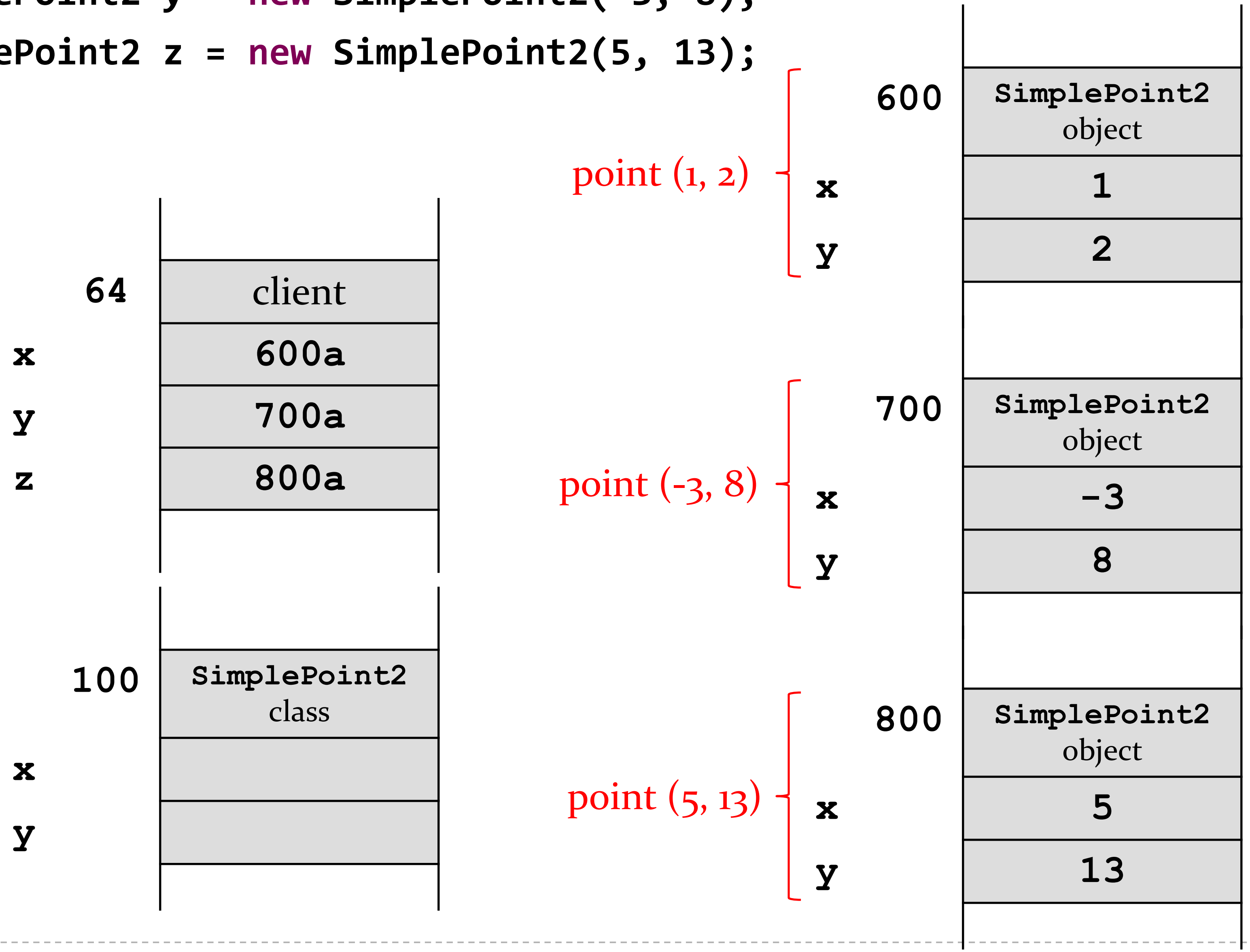
Non-static classes

- a utility class has features (fields and methods) that are all static
  - all features belong to the class
    - therefore, you do not need objects to use those features
      - a well implemented utility class should have a single, empty private constructor to prevent the creation of objects
- most Java classes are *not* utility classes
  - they are intended to be used to create to objects
  - each object has its own copy of all non-static fields
    - it is also useful to imagine that each object has its own copy of all non-static methods

Why objects?

- each object has its own copy of all non-static fields
  - this allows objects to have their own *state*
    - in Java the state of an object is the set of current values of all of its non-static fields
    - e.g., we can create multiple **`SimplePoint2`** objects that all represent different two-dimensional points

```
SimplePoint2 x = new SimplePoint2(1, 2);
SimplePoint2 y = new SimplePoint2(-3, 8);
SimplePoint2 z = new SimplePoint2(5, 13);
```



Implementing classes

- many classes represent kinds of values
  - examples of values: name, date, colour, mathematical point or vector
  - Java examples: **`String, Date, Integer`**
- when implementing a class you need to choose appropriate fields to represent the state of each object
- consider implementing a class that represents 2-dimensional points
  - a possible implementation would have:
    - a field to represent the x-coordinate of the point
    - a field to represent the y-coordinate of the point

```
/**
 * A simple class for representing points in 2D Cartesian
 * coordinates. Every <code>SimplePoint2D</code> instance has a
 * public x and y coordinate that can be directly accessed
 * and modified.
 *
 * @author EECS2030 Winter 2016-17
 *
 */
public class SimplePoint2 {
    public float x;
    public float y;
}
```

`public class`: any client can use this class

`public` fields: any client can use these fields by name

Using **`SimplePoint2`**

- even in its current form, we can use **`SimplePoint2`** to create and manipulate point objects

```java
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2();

  // set its coordinates
  p.x = -1.0f;
  p.y = 1.5f;

  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");
}
```

Using **`SimplePoint2`**

- notice that printing a point is somewhat inconvenient
  - we have to manually compute a string representation of the point
- initializing the coordinates of the point is somewhat inconvenient
  - we have to manually set the x and y coordinates
- we get unusual results when using equals

```java
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2();

  // set its coordinates
  p.x = -1.0f;
  p.y = 1.5f;

  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2();
  q.x = p.x;
  q.y = p.y;

  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

Encapsulation

- we can add features to **`SimplePoint2`** to make it easier to use
  - we can add methods that *use the fields* of **`SimplePoint2`** to perform some sort of computation (like compute a string representation of the point)
  - we can add constructors that *set the values of the fields* of a **`SimplePoint2`** object when it is created
- in object oriented programming the term *encapsulation* means bundling data and methods that use the data into a single unit

Constructors

- the purpose of a constructor is to initialize the state of an object
    - *it should set the values of all of the non-static fields to appropriate values*
- a constructor:
    - must have the same name as the class
    - never returns a value (not even void)
        - constructors are not methods
    - can have zero or more parameters

Default constructor

- the default constructor has zero parameters
- the default constructor initializes the state of an object to some well defined state chosen by the implementer

```
public class SimplePoint2 {
  public float x;
  public float y;

  /**
   * The default constructor. Sets both the x and y coordinate
   * of the point to 0.0f.
   */
  public SimplePoint2() {
    this.x = 0.0f;
    this.y = 0.0f;
  }
```

Inside a constructor, the keyword **`this`** is a reference to the object that is currently being initialized.

Custom constructors

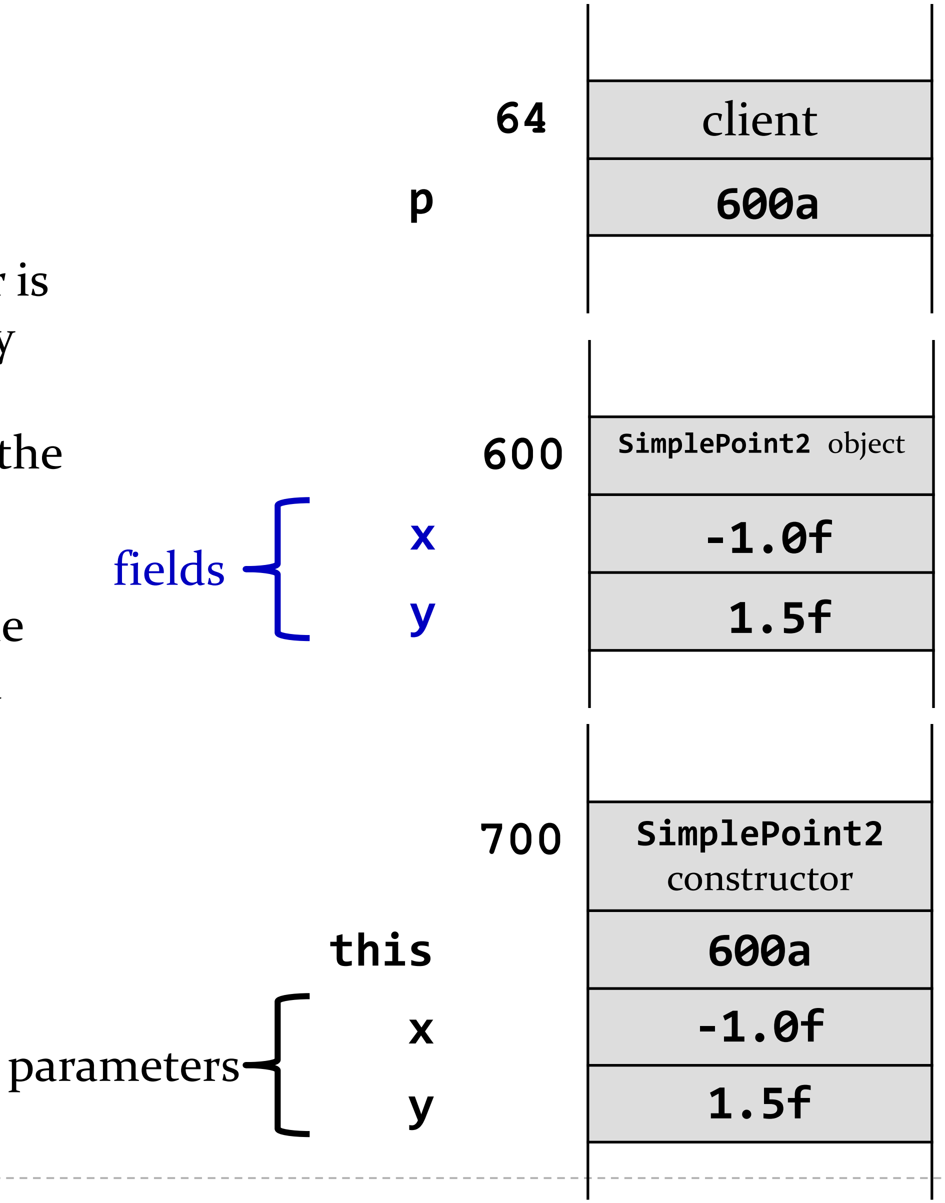
- a class can have multiple constructors but the signatures of the constructors must be unique
  - i.e., each constructor must have a unique list of parameter types
- it would be convenient for clients if **`SimplePoint2`** had a constructor that let the client set the x and y coordinate of the point

```
public class SimplePoint2 {
  public float x;
  public float y;

  /**
   * Sets the x and y coordinate of the point to the argument
   * values.
   *
   * @param x the x coordinate of the point
   * @param y the y coordinate of the point
   */
  public SimplePoint2(float x, float y) {
    this.x = x;
    this.y = y;
  }
```

`this.x` : the field named `x` of `this` point
`this.y` : the field named `y` of `this` point
`x` : the parameter named `x` of the constructor
`y` : the parameter named `y` of the constructor

`SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);`

1. **`new`** allocates memory for a **`SimplePoint2`** object
2. the **`SimplePoint2`** constructor is invoked by passing the memory address of the object and the arguments **`-1.0f`** and **`1.5f`** to the constructor
3. the constructor runs, setting the values of the fields **`this.x`** and **`this.y`**
4. the value of **`p`** is set to the memory address of the constructed object



64 client
p 600a

600 SimplePoint2 object
fields
x -1.0f
y 1.5f

700 SimplePoint2 constructor
this 600a
parameters
x -1.0f
y 1.5f

`this`

- in our constructor

```
public SimplePoint2(float x, float y) {
    this.x = x;
    this.y = y;
 }
```

there are parameters with the same names as fields

- when this occurs, the parameter has precedence over the field
    - we say that the parameter *shadows* the field
    - when shadowing occurs you must use **`this`** to refer to the field

Custom constructors

- adding the constructor `SimplePoint2(float x, float y)` allows the client to simplify their code

```java
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2();

  // set its coordinates
  p.x = -1.0f;
  p.y = 1.5f;

  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2();
  q.x = p.x;
  q.y = p.y;

  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);



  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2(p.x, p.y);



  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

Copy constructor

- a copy constructor initializes the state of an object by copying the state of another object (having the same type)
  - it has a single parameter that is the same type as the class

```
public class SimplePoint2 {
  public float x;
  public float y;


  /**
   * Sets the x and y coordinate of this point by copying
   * the x and y coordinate of another point.
   *
   * @param other a point to copy
   */
  public SimplePoint2(SimplePoint2 other) {
    this.x = other.x;
    this.y = other.y;
  }
```

Copy constructor

- adding a copy constructor allows the client to simplify their code

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);






  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2(~~p.x, p.y~~);




  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);






  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2(p);




  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

Avoiding Code Duplication

- notice that the constructor bodies are almost identical to each other
  - all three constructors have 2 lines of code
  - all three constructors set the x and y coordinate of the point

- whenever you see duplicated code you should consider moving the duplicated code into a method
- in this case, one of the constructors already does everything we need to implement the other constructors…

Constructor chaining

- a constructor is allowed to invoke another constructor
- 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
    - but you *cannot* use **`this`** in a method to invoke a constructor
- we can re-write two of our constructors to use constructor chaining...

```
public class SimplePoint2 {
  public float x;
  public float y;

  public SimplePoint2() {
    this(0.0f, 0.0f);                              invokes
  }

  public SimplePoint2(float x, float y) {
    this.x = x;
    this.y = y;
  }

  public SimplePoint2(SimplePoint2 other) {
    this(other.x, other.y);                          invokes
  }
```

Methods

- a method performs some kind of computation
- a *non-static* method can use any field belonging to an object in the computation
- for example, we can provide a non-static method that allows the client to set both the x and y coordinates of the point

```java
/**
 * Sets the x and y coordinate of this point to the argument
 * values.
 *
 * @param x the new x coordinate of the point
 * @param y the new y coordinate of the point
 */
public void set(float x, float y) {
  this.x = x;
  this.y = y;
}
```

Obligatory methods

- in Java every class is actually a child class of the class **`java.lang.Object`**
  - this means that every class has methods that it inherits from **`java.lang.Object`**
    - there are 11 such methods, but 3 are especially important to us:
      - ☐ **`toString`**
      - ☐ **`equals`**
      - ☐ **`hashCode`**

`toString`

- the **`toString`** method should return a textual representation of the object
- a textual representation of the point **`p`**

```
SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);
```

might be something like **`(-1.0, 1.5)`**

```
/**
 * Returns a string representation of this point. The string
 * representation of this point is the x and y-coordinates
 * of this point, separated by a comma and space, inside a pair
 * of parentheses.
 *
 * @return a string representation of this point
 */
@Override
public String toString() {
  return "(" + this.x + ", " + this.y + ")";
}
```

`@Override` is an optional annotation that we can use to tell the compiler that we are redefining the behavior of the **`toString`** method that was inherited from **`java.lang.Object`**

`toString`

- by providing **`toString`** clients can now easily get a string representation of a **`SimplePoint2`** object

```java
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);






  // get its coordinates
  System.out.println("p = ~~(" + p.x + ", " + p.y + ")~~");

  SimplePoint2 q = new SimplePoint2(p);




  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);







  // get its coordinates
  System.out.println("p = " + p.toString());

  SimplePoint2 q = new SimplePoint2(p);




  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

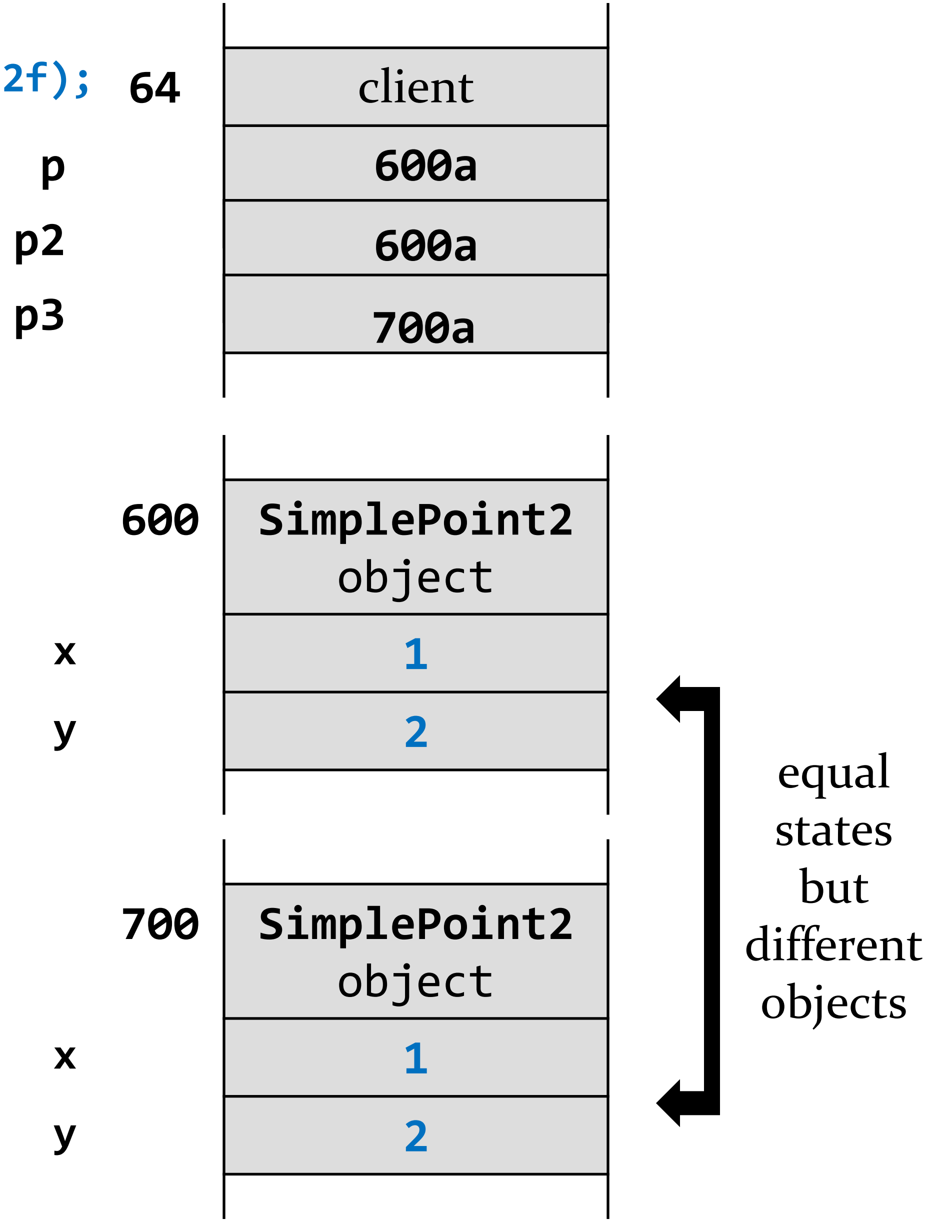
`equals`

- suppose you write a value class that extends **`Object`** but you do not override **`equals()`**
  - what happens when a client tries to use **`equals()`**?
    - **`Object.equals()`** is called

```
// SimplePoint2 client

SimplePoint2 p = new SimplePoint2(1f, 2f);
System.out.println( p.equals(p) );        // true

SimplePoint2 p2 = p;
System.out.println( p2.equals(p) );       // true

SimplePoint2 p3 = new SimplePoint2(1f, 2f);
System.out.println( p3.equals(p));        // false!
```

[notes 3.2.4]

```
SimplePoint2 p = new SimplePoint2(1f, 2f);
SimplePoint2 p2 = p;
SimplePoint2 p3 = new SimplePoint2(1f, 2f);
```



p and **p2** refer to the object at address 600

p3 refers to the object at address 700

equal states but different objects

`Object.equals`

- **`Object.equals`** checks if two references refer to the same object
  - **`x.equals(y)`** is true if and only if **`x`** and **`y`** are references to the same object

`SimplePoint2.equals`

- most value classes should support logical equality
  - an instance is equal to another instance if their states are equal
    - e.g. two points are equal if their x and y coordinates both have the same values

- implementing **`equals()`** is surprisingly hard
  - "One would expect that overriding **`equals()`**, since it is a fairly common task, should be a piece of cake. The reality is far from that. There is an amazing amount of disagreement in the Java community regarding correct implementation of **`equals()`**. Look into the best Java source code or open an arbitrary Java textbook and take a look at what you find. Chances are good that you will find several different approaches and a variety of recommendations."
    - Angelika Langer, Secrets of equals() – Part 1
  - `http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html`

- what we are about to do does not always produce the result you might be looking for
  - but it is always satisfies the **`equals()`** contract
  - and it's what the notes and textbook do

EECS2030 Requirements for `equals`

1. an instance is equal to itself
2. an instance is never equal to `null`
3. only instances of the exact same type can be equal
4. instances with the same state are equal

1. An Instance is Equal to Itself

- **`x.equals(x)`** should always be **`true`**
- also, **`x.equals(y)`** should always be true if **`x`** and **`y`** are references to the same object
- you can check if two references are equal using **`==`**

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

}
```

2. An Instance is Never Equal to `null`

- Java requires that `x.equals(null)` returns `false`
- and you must not throw an exception if the argument is `null`
    - so it looks like we have to check for a `null` argument…

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


}
```

3. Instances of the same type can be equal

- the implementation of **`equals()`** used in the notes and the textbook is based on the rule that an instance can only be equal to another instance of the same type
- you can find the class of an object using **`Object.getClass()`**

```
public final Class<? extends Object> getClass()
```

Returns the runtime class of an object.

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


}
```

Instances with Same State are Equal

- recall that the value of the fields of an object define the state of the object
  - two instances are equal if all of their fields are equal

- unfortunately, we cannot yet retrieve the attributes of the parameter **`obj`** because it is declared to be an **`Object`** in the method signature
  - we need a cast

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  SimplePoint2 other = (SimplePoint2) obj;



}
```

Instances with Same State are Equal

- there is a recipe for checking equality of fields

1. if the field is a primitive type other than **`float`** or **`double`** use **`==`**
2. if the field type is **`float`** use **`Float.floatToLongBits`**
3. if the attribute type is **`double`** use **`Double.doubleToLongBits`**
4. if the field is an array consider **`Arrays.equals`**
5. if the field is a reference type use **`equals`**, but beware of fields that might be null

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  SimplePoint2 other = (SimplePoint2) obj;
  if (Float.floatToIntBits(this.x) != Float.floatToIntBits(other.x)) {
    return false;
  }
  if (Float.floatToIntBits(this.y) != Float.floatToIntBits(other.y)) {
    return false;
  }
  return true;
}
```

`equals`

- our version of **`equals`** compares the state of two points to determine equality
  - now two points with the same coordinates are considered equal

```java
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);



  // get its coordinates
  System.out.println("p = " + p.toString());

  SimplePoint2 q = new SimplePoint2(p);



  // equals? yes!
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

true

The `equals` Contract

- for reference values `equals` is
  1. reflexive
  2. symmetric
  3. transitive
  4. consistent
  5. must not throw an exception when passed `null`

The **`equals`** contract: Reflexivity

1. reflexive :
    - an object is equal to itself
    - **`x.equals(x)`** is **`true`**

The **`equals`** contract: Symmetry

2. symmetric :
   - two objects must agree on whether they are equal
   - **`x.equals(y)`** is **`true`** if and only if **`y.equals(x)`** is **`true`**

The `equals` contract: Transitivity

3. transitive :
   - if a first object is equal to a second, and the second object is equal to a third, then the first object must be equal to the third
   - if
     `x.equals(y)` is **`true`**
     and
     `y.equals(z)` is **`true`**
     then
     `x.equals(z)` must be **`true`**

The `equals` contract: Consistency

4. consistent :
    - repeatedly comparing two objects yields the same result (assuming the state of the objects does not change)

The **`equals`** contract: Non-nullity

5. **`x.equals(null)`** is always **`false`** and never throws an exception

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 **`SimplePoint2`**

```
// client code somewhere
SimplePoint2 p = new SimplePoint2(1f, -2f);

HashSet<SimplePoint2> h = new HashSet<>();
h.add(p);
System.out.println( h.contains(p) );       // true

SimplePoint2 q = new SimplePoint2(1f, -2f);
System.out.println( h.contains(q) );       // false!
```

[notes 3.3.5]

Arrays as Containers

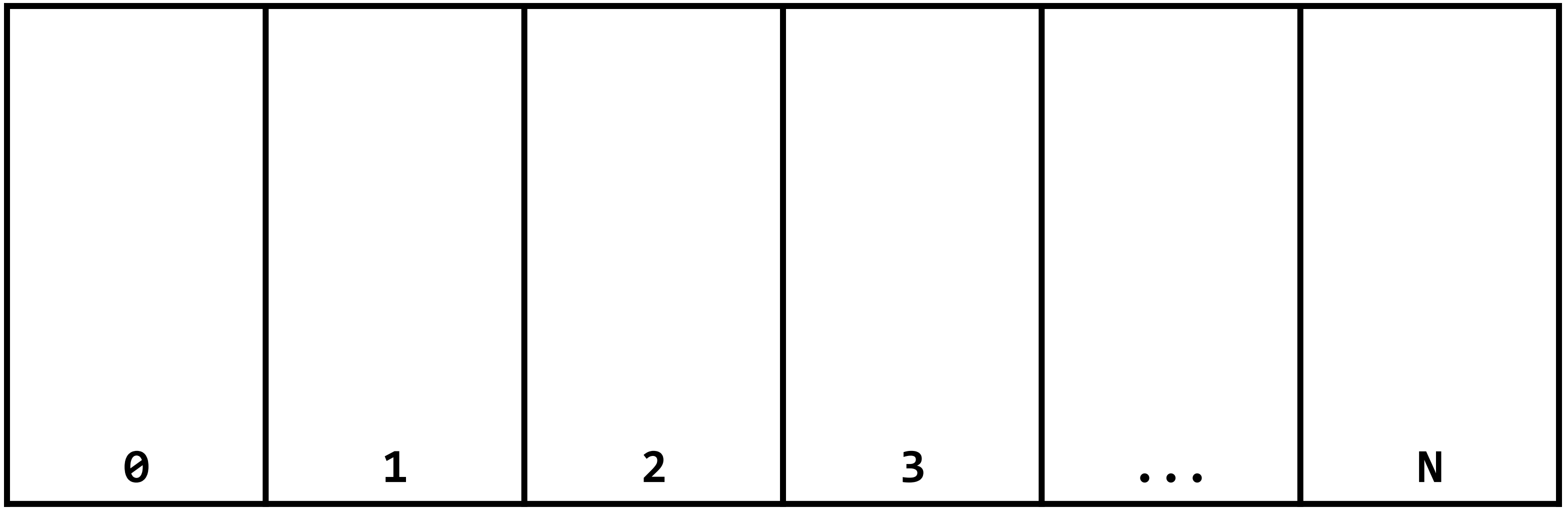
- suppose you have a list of unique **`SimplePoint2`** points
  - how do you compute whether or not the list contains a particular point?
    - write a loop to examine every element of the list

```
public static boolean
       hasPoint(SimplePoint2 p, List<SimplePoint2> points) {

  for( SimplePoint2 point : points ) {
    if (point.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`** elements in the list:
  - best case
    - the first element is the one we are searching for
      - 1 call to **`equals`**
  - worst case
    - the element is not in the list
      - n calls to **`equals`**
  - average case
    - the element is somewhere in the middle of the list
      - approximately **`(n/2)`** calls to **`equals`**

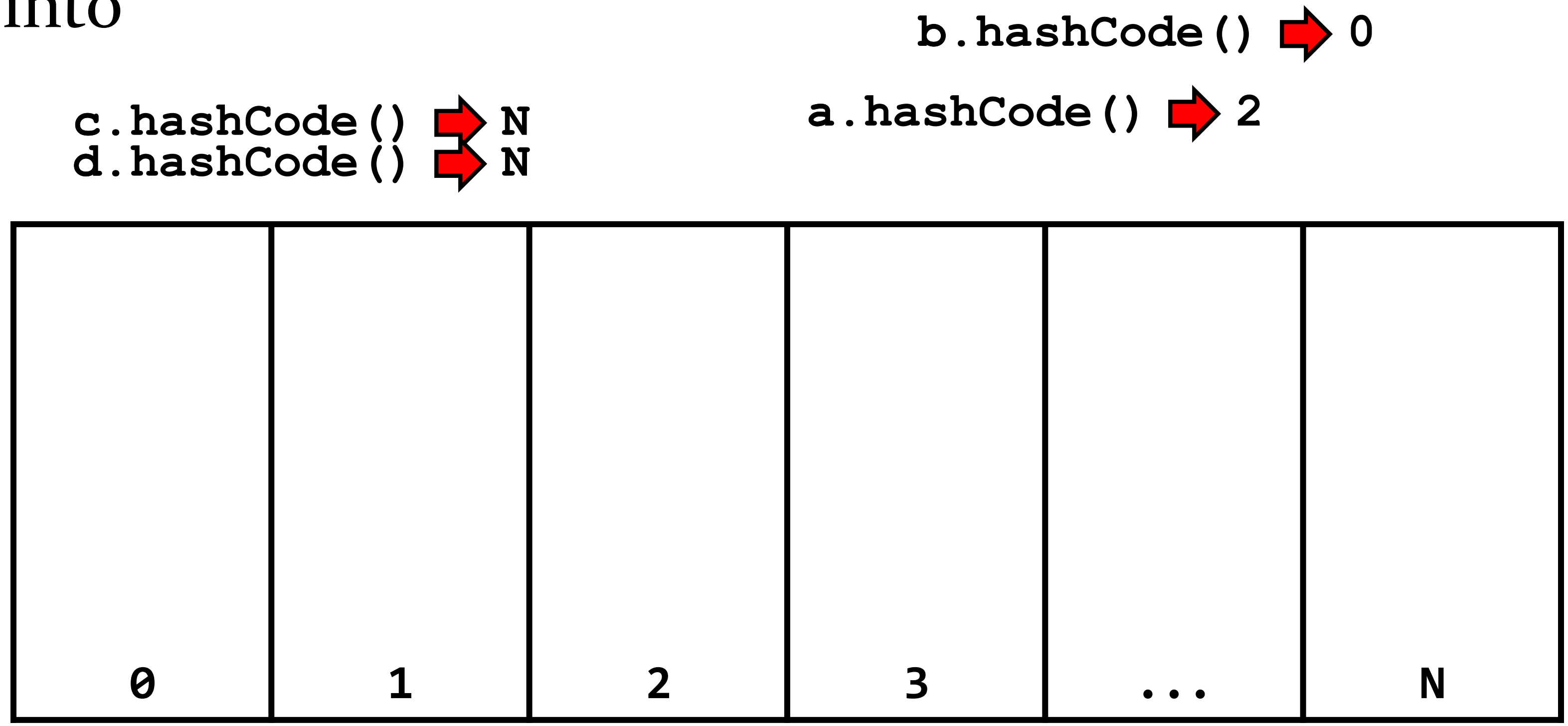
Hash Tables

- 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



`b.hashCode()` ➡ `0`

`a.hashCode()` ➡ `2`

`c.hashCode()` ➡ `N`
`d.hashCode()` ➡ `N`

`0`	`1`	`2`	`3`	`...`	`N`

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

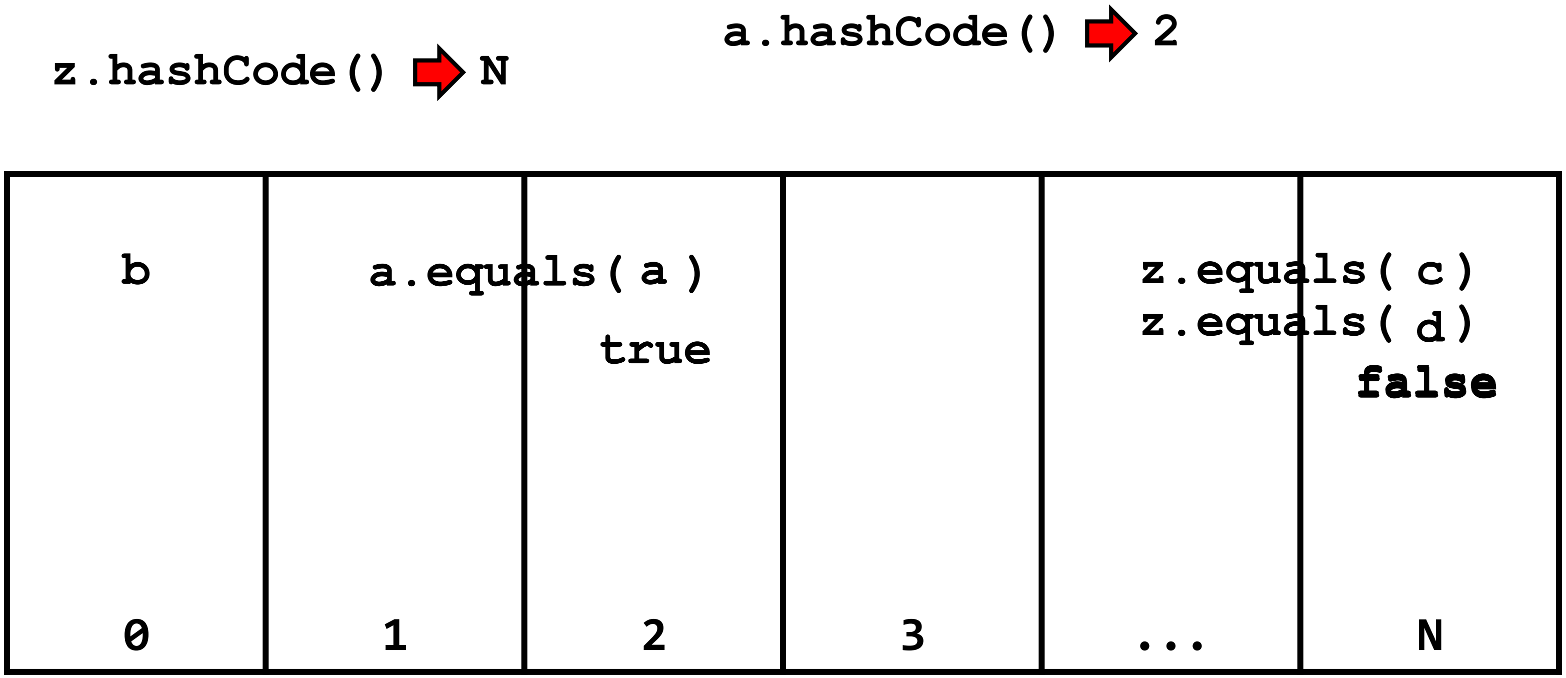
Insertion into a Hash Table

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

`b`		`a`			`c` `d`
`0`	`1`	`2`	`3`	`...`	`N`

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



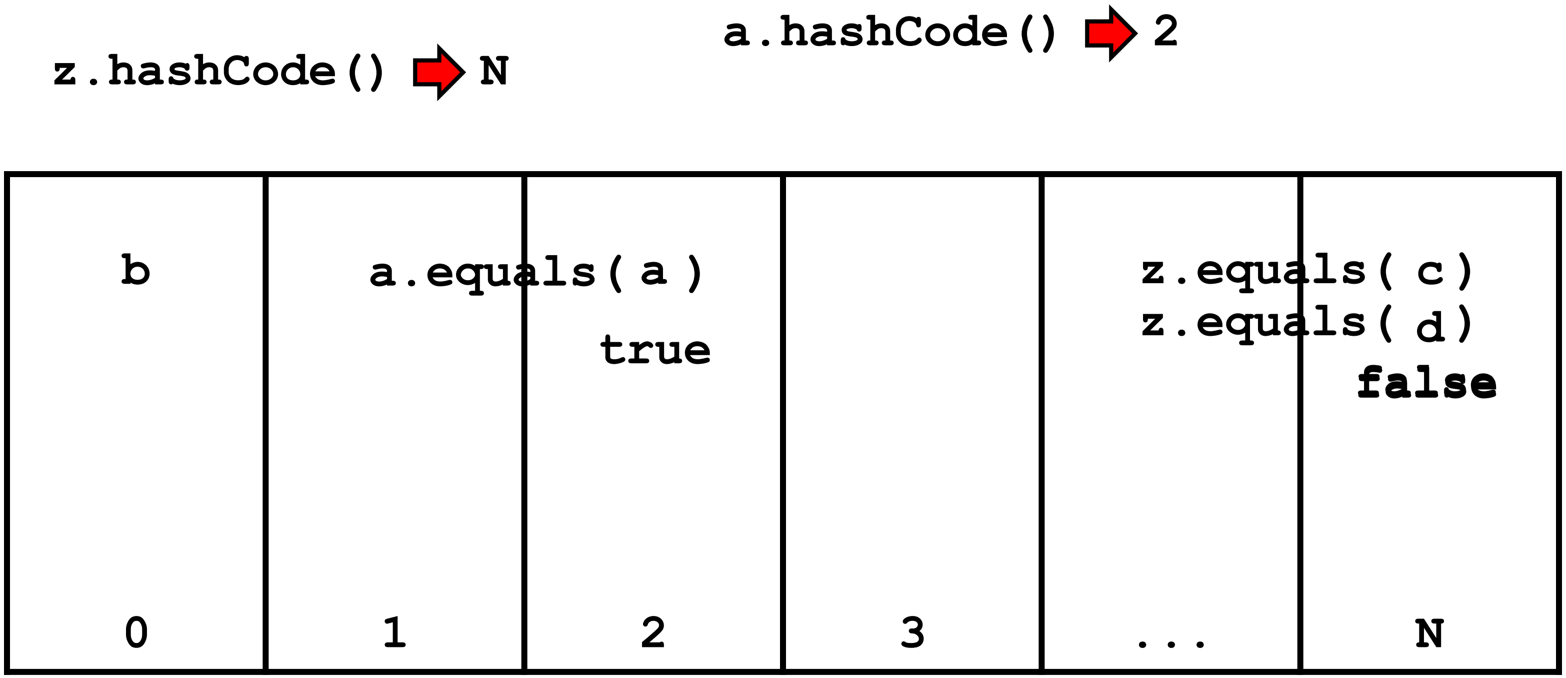
`a.hashCode()` ➡ 2

`z.hashCode()` ➡ N

0	1	2	3	...	N
b	a.equals(a)	true		z.equals(c) z.equals(d)	false

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



`a.hashCode()` ➡ 2

`z.hashCode()` ➡ N

0	1	2	3	...	N
b	a.equals(a)	true		z.equals(c) z.equals(d)	false

- 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`** elements in the hash table:
  - best case
    - the bucket is empty, or the first element in the bucket is the one we are searching for
      - ☐ 0 or 1 call to **`equals`**
  - worst case
    - all **`n`** of the elements are in the same bucket
      - ☐ n calls to **`equals`**
  - average case
    - the element is in a bucket with a small number of other elements
      - ☐ 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

```
// client code somewhere
SimplePoint2 p = new SimplePoint2(1f, -2f);

HashSet<SimplePoint2> h = new HashSet<>();
h.add(p);
System.out.println( h.contains(p) );       // true

SimplePoint2 q = new SimplePoint2(1f, -2f);
System.out.println( h.contains(q) );       // false!
```

- note that **`p`** and **`q`** refer to 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 complex number even though **`p.equals(q)`** is **`true`**

Implementing **hashCode**

- the basic idea is generate a hash code using the fields 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

A bad (but legal) **hashCode**

```
public class SimplePoint2 {
  public float x;
  public float y;

  @Override
  public int hashCode() {
    return 1;
  }
```

- this will cause a hashed container to put all points into the same bucket

A slightly better **hashCode**

```
public class SimplePoint2 {
  public float x;
  public float y;

  @Override
  public int hashCode() {
    return (int) (this.x + this.y);
  }
```

A good **hashCode**

```
public class SimplePoint2x {
  public float x;
  public float y;

  @Override
  public int hashCode() {
    return Objects.hash(this.x, this.y);
  }
```

eclipse **hashCode**

- eclipse will also generate a hashCode method for you
  - Source → Generate hashCode() and equals()…

- it uses an algorithm that
  - "… yields reasonably good hash functions, [but] does not yield state-of-the-art hash functions, nor do the Java platform libraries provide such hash functions as of release 1.6. Writing such hash functions is a research topic, best left to mathematicians and theoretical computer scientists."
    - Joshua Bloch, *Effective Java 2nd Edition*