Object Tools

CPSC 233: Introduction to Computer Science for Computer Science Majors II
Winter 2022

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

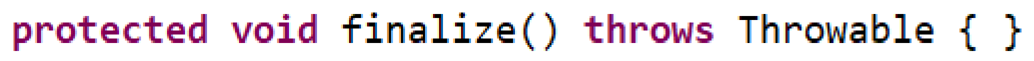
Wednesday, 10 November 2021

Objects

- Every class is extended from the Object class
- What do we inherit?
  - protected void finalize()
  - public final Class getClass()
  - public String toString()
  - protected Object clone()
  - public Boolean equals(Object obj)
  - public int hashCode()

public void finalize()

Lets you cleanup when an object is garbage collected (deallocating memory happens for free unlike C, but sometimes you have other attachments to cleanup)



```
protected void finalize() throws Throwable { }
```

Doesn’t do anything in Object as there is nothing to clean-up

public void finalize()

We can override this if we think we have special things to clean-up

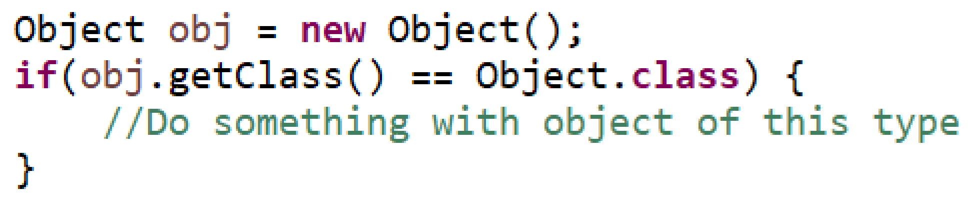
However, garbage collection is not under our control

- You can attempt to trigger it sooner with System.gc() but still no guarantees

Being phased out with Java 9 and better ways to cleanup after ourselves like AutoCloseable interface

public final Class getClass()

Returns the runtime class of this Object.



```
Object obj = new Object();
if(obj.getClass() == Object.class) {
    //Do something with object of this type
}
```

public final Class getClass()

Similar to instanceof except it is a strict comparison

i.e. if B extends A, then

A instanceof A is true and A.getClass() == A.class is true

B instanceof A is true but B.getClass() == A.class is false

public String toString()

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

getClass().getName() is the String version of the class name

- This String includes the package path of the Object

```
System.out.println((new Object()).getClass().getName());
```

- Returns “java.lang.Object”

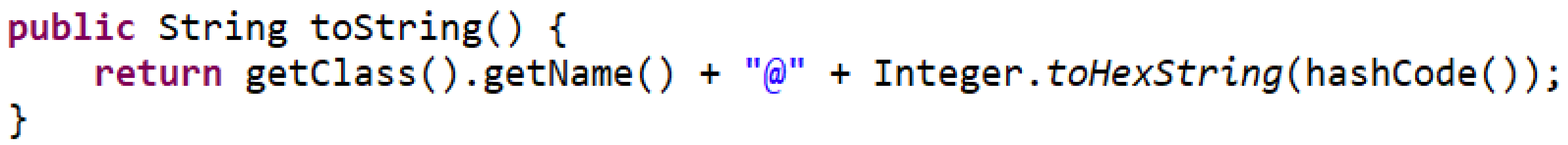
public String toString()

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

getClass().getName() is the String version of the class name

- But, what is hashCode() as a hexString(), for now it is just a hexadecimal [0-9a-f] string of internal integer id given to each object as it is created in Java

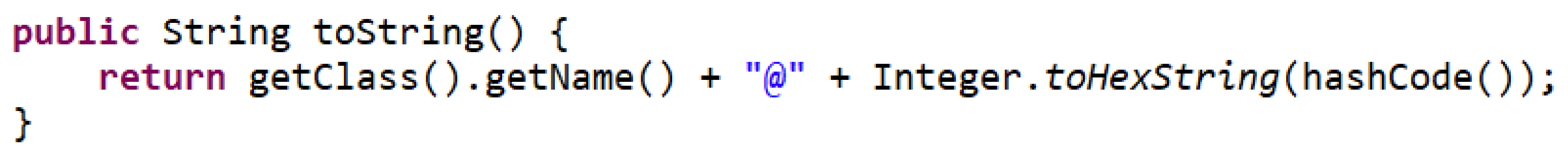
public String toString()



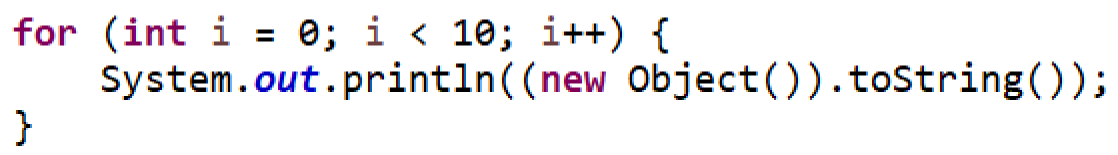
```java
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- We have already had many cases where we used our ability to override this method

public String toString()



```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```



```
for (int i = 0; i < 10; i++) {
    System.out.println((new Object()).toString());
}
```

- java.lang.Object@7852e922
- java.lang.Object@4e25154f
- java.lang.Object@70dea4e
- java.lang.Object@5c647e05
- java.lang.Object@33909752
- java.lang.Object@55f96302
- java.lang.Object@3d4eac69
- java.lang.Object@42a57993

public Object clone()

Clone (Copy) an object (unclear what type it has to be!)

Two types of copying exist

- Shallow copy
- Deep copy

public Object clone()

Shallow copy

- all instance primitive type variables have the same number
- all instance object pointers point to the same object

public Object clone()

Deep copy

- all instance primitive type variables have the same number
- all instance object pointer point to a new deep copy of those objects

public Object clone()

Shallow copy

- all instance base type variables have the same number
- all instance object pointer point to the same object

Deep copy (PREFERRED)

- all instance base type variables have the same number
- all instance object pointer point to a new deep copy of the object

public Object clone()

Shallow copy

- all instance base type variables have the same number
- all instance object pointer point to the same object

•(WHY NOT? -> unintended consequences)

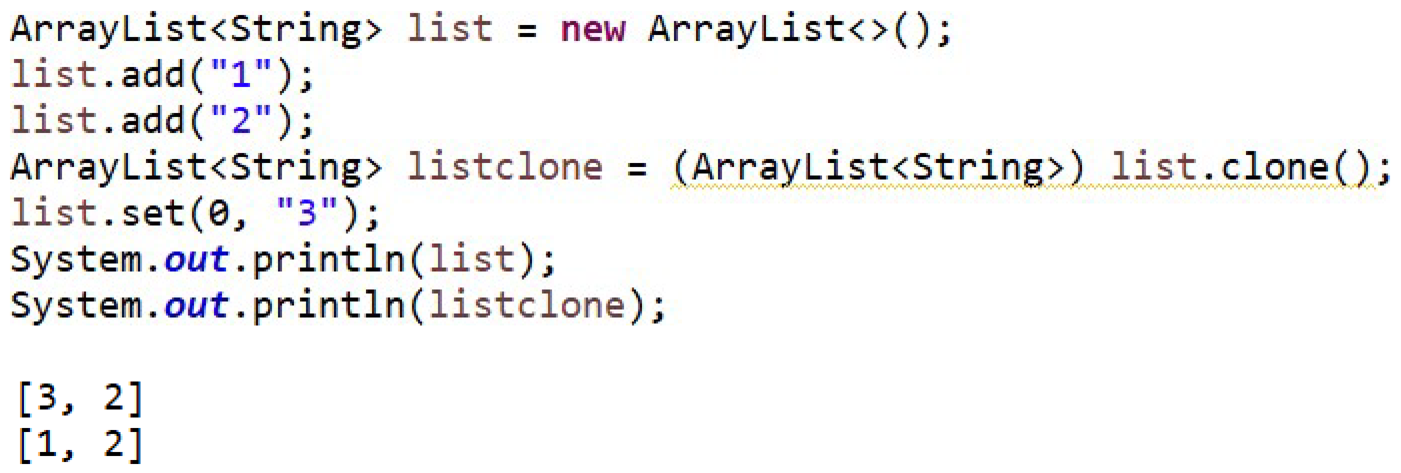
•Deep copy (PREFERRED)

- all instance base type variables have the same number
- all instance object pointer point to a new deep copy of the object

public Object clone()

To start clone can make our life easier with a simple list of Java primitives like String

Nice easy copy



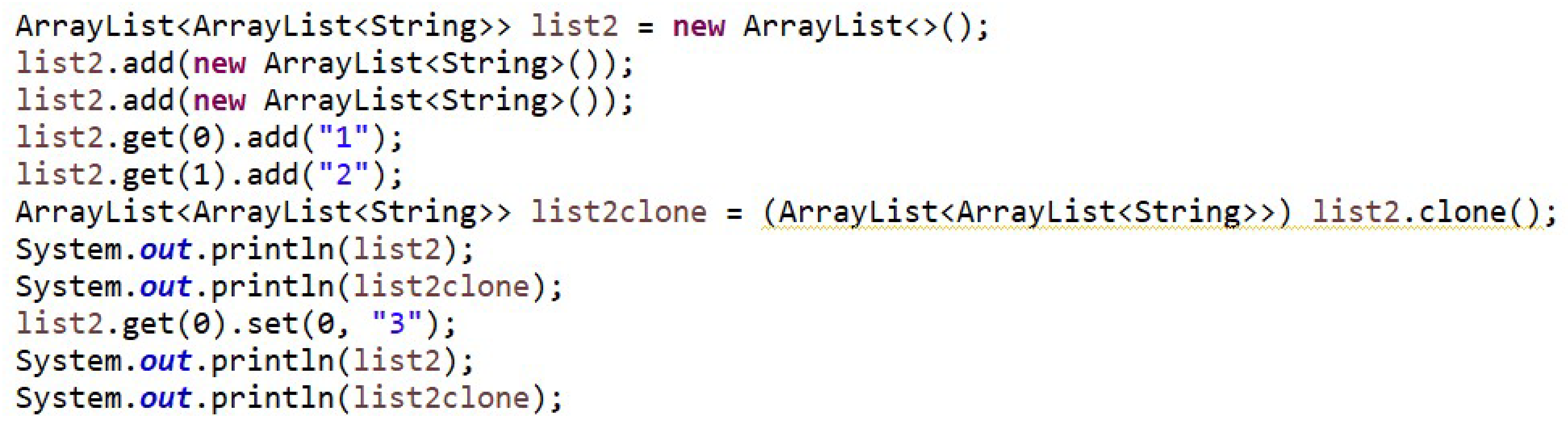
```
ArrayList<String> list = new ArrayList<>();
list.add("1");
list.add("2");
ArrayList<String> listclone = (ArrayList<String>) list.clone();
list.set(0, "3");
System.out.println(list);
System.out.println(listclone);
```

```
[3, 2]
[1, 2]
```

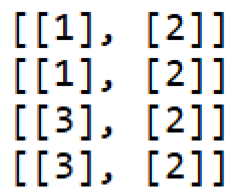
public Object clone()

To start clone can make our life easier with a simple list of Java primitives like String

But the ArrayList clone is shallow



```java
ArrayList<ArrayList<String>> list2 = new ArrayList<>();
list2.add(new ArrayList<String>());
list2.add(new ArrayList<String>());
list2.get(0).add("1");
list2.get(1).add("2");
ArrayList<ArrayList<String>> list2clone = (ArrayList<ArrayList<String>>) list2.clone();
System.out.println(list2);
System.out.println(list2clone);
list2.get(0).set(0, "3");
System.out.println(list2);
System.out.println(list2clone);
```



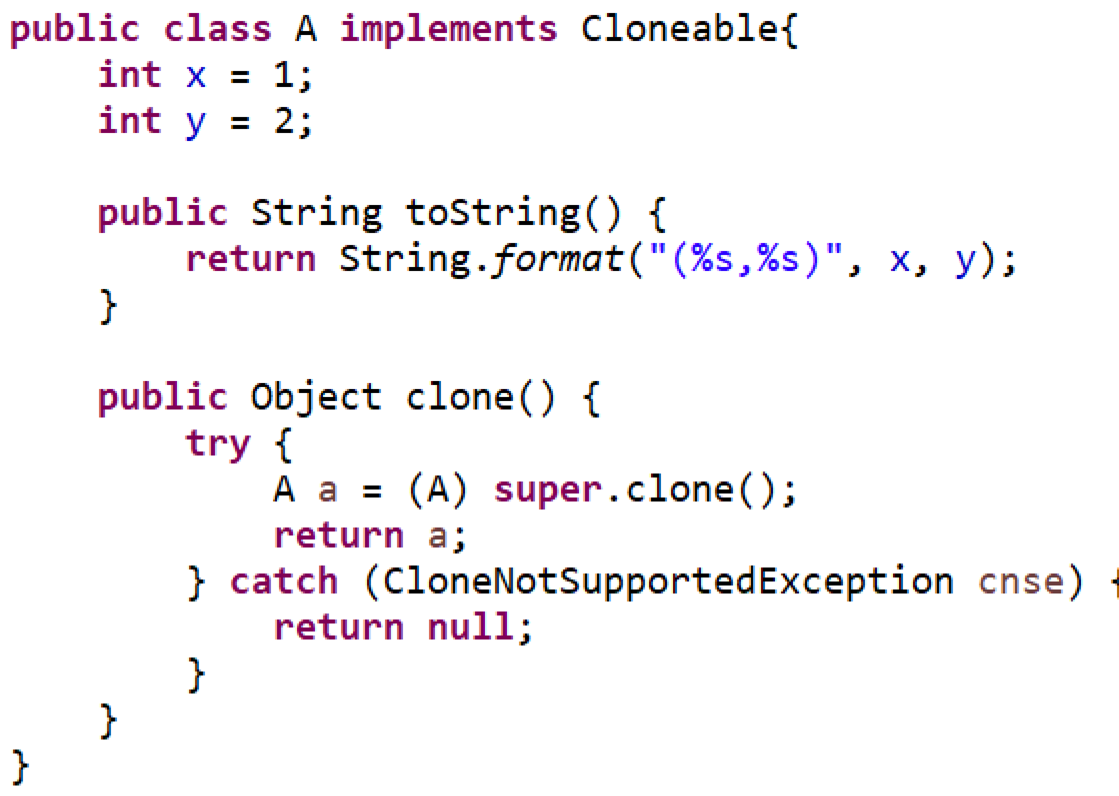
```
[[1], [2]]
[[1], [2]]
[[3], [2]]
[[3], [2]]
```

public Object clone()

How do we implement a Cloneable class

- In Java 8
- Implement Cloneable
- Override clone()
- Have to try-catch wrap exception that is thrown
- If we just have primitive types we can use Java's

super.clone() to handle things for us



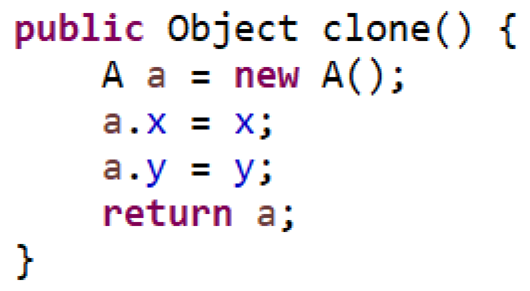
```java
public class A implements Cloneable{
    int x = 1;
    int y = 2;

    public String toString() {
        return String.format("(%s,%s)", x, y);
    }

    public Object clone() {
        try {
            A a = (A) super.clone();
            return a;
        } catch (CloneNotSupportedException cnse)
            return null;
        }
    }
}
```

public Object clone()

Alternative is to create a new object

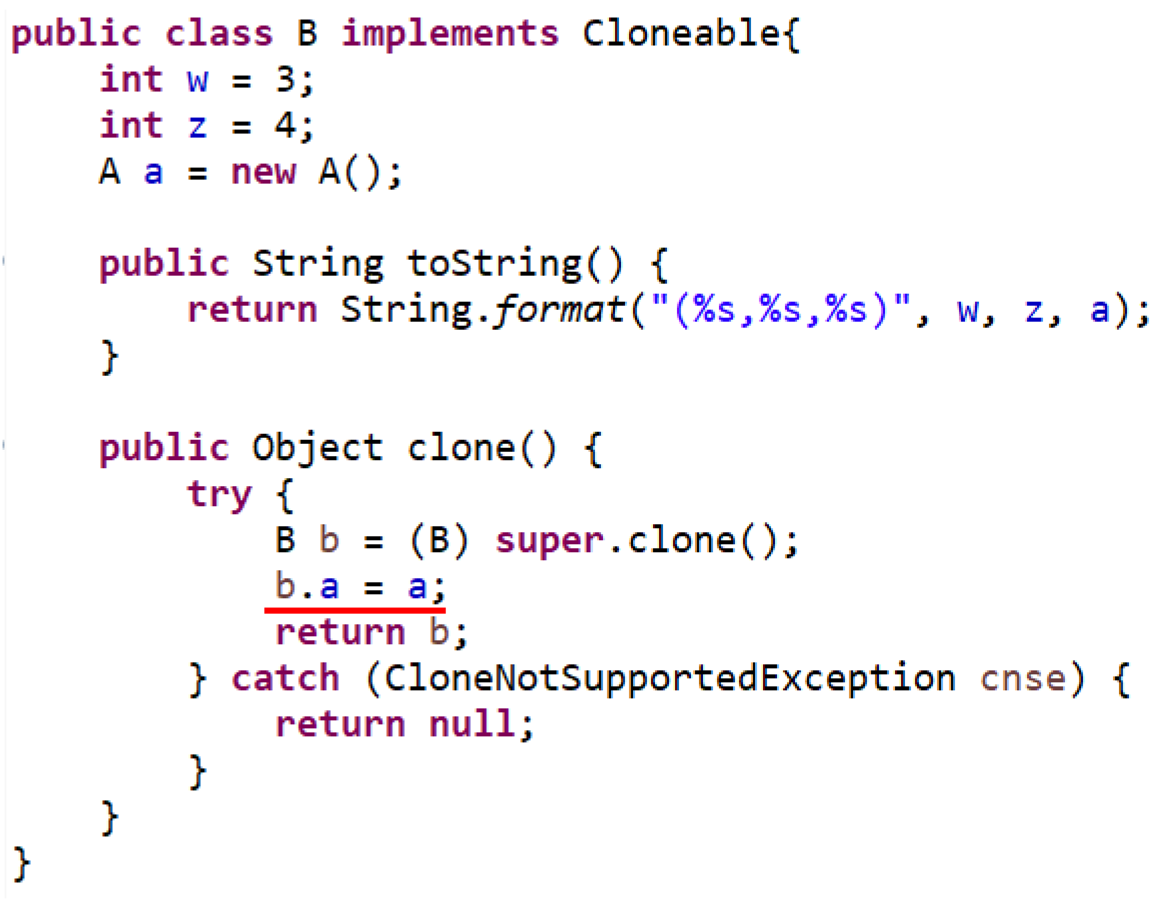


```
public Object clone() {
    A a = new A();
    a.x = x;
    a.y = y;
    return a;
}
```

Easier to do

public Object clone()

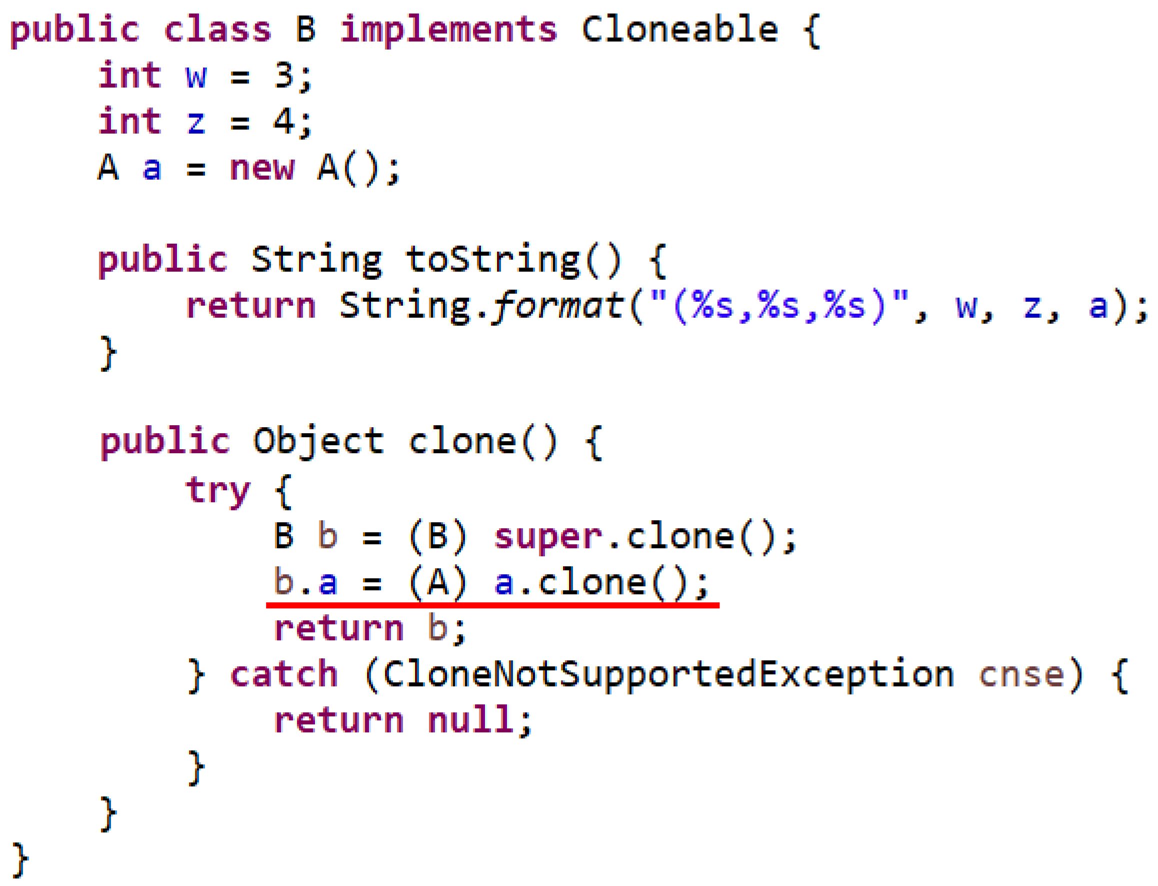
Shallow vs deep



```java
public class B implements Cloneable{
    int w = 3;
    int z = 4;
    A a = new A();

    public String toString() {
        return String.format("(%s,%s,%s)", w, z, a);
    }

    public Object clone() {
        try {
            B b = (B) super.clone();
            b.a = a;
            return b;
        } catch (CloneNotSupportedException cnse) {
            return null;
        }
    }
}
```



```java
public class B implements Cloneable {
    int w = 3;
    int z = 4;
    A a = new A();

    public String toString() {
        return String.format("(%s,%s,%s)", w, z, a);
    }

    public Object clone() {
        try {
            B b = (B) super.clone();
            b.a = (A) a.clone();
            return b;
        } catch (CloneNotSupportedException cnse) {
            return null;
        }
    }
}
```

public Object clone()

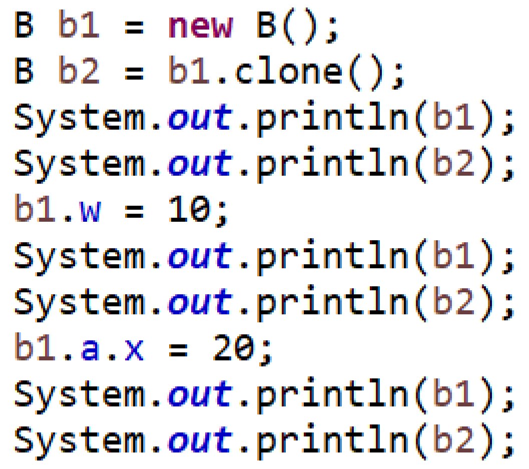
Shallow vs Deep

```
B b1 = new B();
B b2 = b1.clone();
System.out.println(b1);
System.out.println(b2);
b1.w = 10;
System.out.println(b1);
System.out.println(b2);
b1.a.x = 20;
System.out.println(b1);
System.out.println(b2);
```

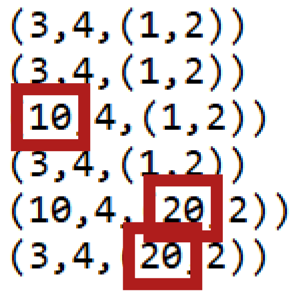
```
(3,4,(1,2))        (3,4,(1,2))
(3,4,(1,2))        (3,4,(1,2))
(10,4,(1,2))       (10,4,(1,2))
(3,4,(1,2))        (3,4,(1,2))
(10,4,(20,2))      (10,4,(20,2))
(3,4,(20,2))       (3,4,(1,2))
```

public Object clone()

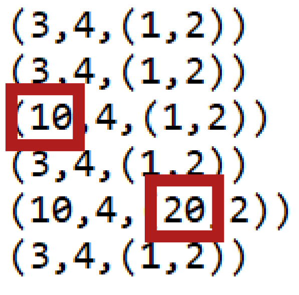
Shallow vs Deep



```java
B b1 = new B();
B b2 = b1.clone();
System.out.println(b1);
System.out.println(b2);
b1.w = 10;
System.out.println(b1);
System.out.println(b2);
b1.a.x = 20;
System.out.println(b1);
System.out.println(b2);
```



```
(3,4,(1,2))
(3,4,(1,2))
(10,4,(1,2))
(3,4,(1,2))
(10,4,(20,2))
(3,4,(20,2))
```



```
(3,4,(1,2))
(3,4,(1,2))
(10,4,(1,2))
(3,4,(1,2))
(10,4,(20,2))
(3,4,(1,2))
```

public Object clone()

Simple Shallow vs Deep

```
public Object clone() {
     B b = new B();
     b.w = w;
     b.z = z;
     b.a = a;
     return b;
 }
```

```
public Object clone() {
    B b = new B();
    b.w = w;
    b.z = z;
    b.a = (A) a.clone();
    return b;
}
```

public int hashCode()

This was introduced before (use for storing objects in some Collections)

By default this is the object’s creation internal integer id

These are unique for each object created

A hashCode should be

- The same every time it is used (unless object contents have changed)
- Two equal objects should produce the same hashCode

Desirable but required to guarantee that

- If two objects are **not equal** that their hashCode is different

public boolean equals(Object obj)

Is one object equal to another.

Where have we seen it before?

String usage

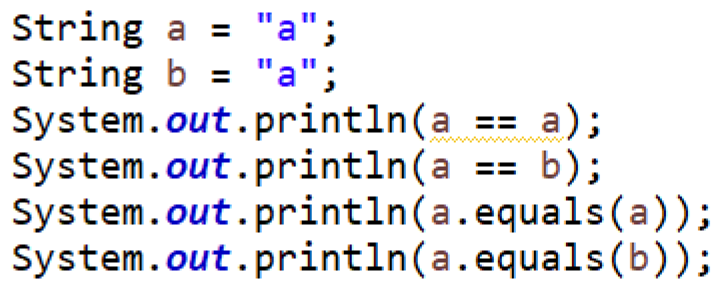
You’ve been recommended to use

“string content”.equals(“string content”)

But why?

public boolean equals(Object obj)

Because == is only guaranteed to compare if two object variables point at the same object (not if the contents of the object are the same)

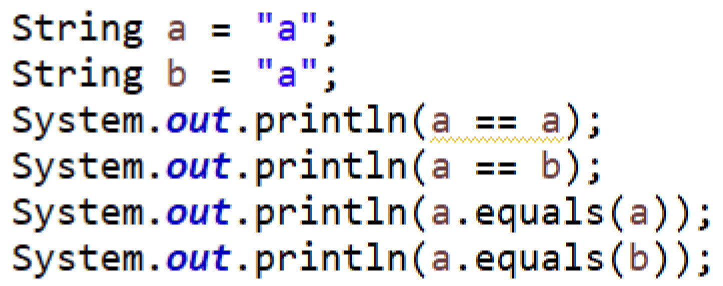


```
String a = "a";
String b = "a";
System.out.println(a == a);
System.out.println(a == b);
System.out.println(a.equals(a));
System.out.println(a.equals(b));
```

a == b may return false

public boolean equals(Object obj)

Because == is only guaranteed to compare if two object variables point at the same object (not if the contents of the object are the same)



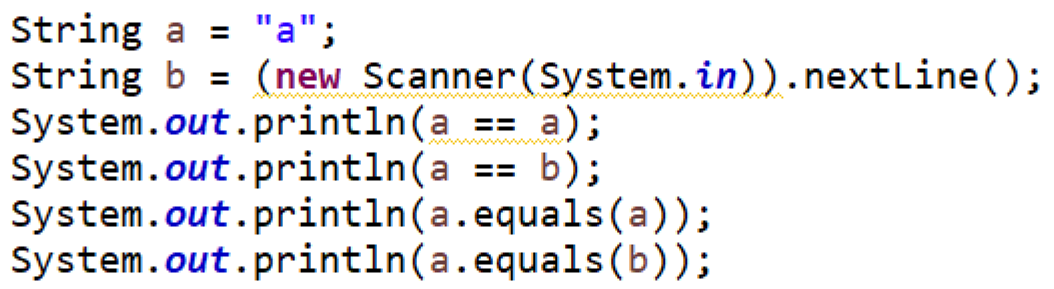
```
String a = "a";
String b = "a";
System.out.println(a == a);
System.out.println(a == b);
System.out.println(a.equals(a));
System.out.println(a.equals(b));
```

a == b may return false (Why is this may?)

Java actually does something called interning for String since they are immutable. If it sees a String already used it will try to re-use the same object again. But no guarantees.

public boolean equals(Object obj)

Example, it is more likely a==b is false because the String read in will not get chance for intern

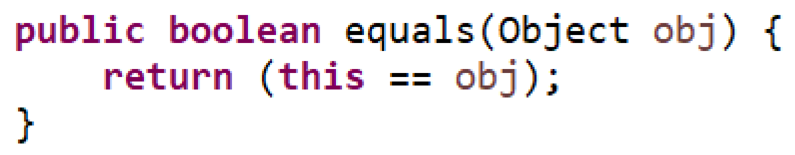


```
String a = "a";
String b = (new Scanner(System.in)).nextLine();
System.out.println(a == a);
System.out.println(a == b);
System.out.println(a.equals(a));
System.out.println(a.equals(b));
```

In summary, for String equals is a safe String comparison while == is not safe to guarantee the content of two strings are equal

public boolean equals(Object obj)

Default equals



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

It is generally accepted that if you plan to use java.util.Collections and you override equals that you also override hashCode

Also it is generally accepted that if this.equals(other) then this.compare(other) == 0

public boolean equals(Object obj)

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.

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

public boolean equals(Object obj)

We can make our own

The keys to external definition

1. public
2. boolean
3. equals
4. Must be Object -> if you have a different class here such as the name of the class the equals function is in you haven’t overridden equals you’ve created an equals with a different signature

public boolean equals(Object obj)

We can make our own

The keys to external definition

1. public
2. boolean
3. equals
4. Must be Object (Most Common Error) -> if you have a different class here such as the name of the class the equals function is in you haven’t overridden equals you’ve created an equals with a different signature

public boolean equals(Object obj)

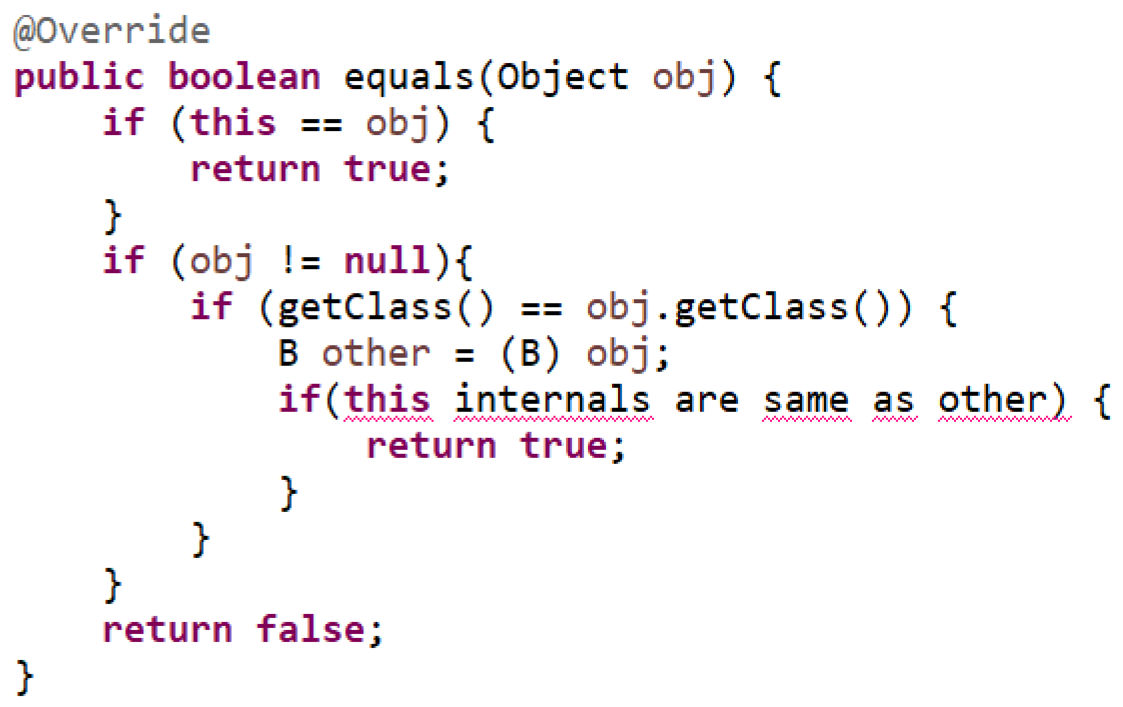
We can make our own

The keys to internal definition

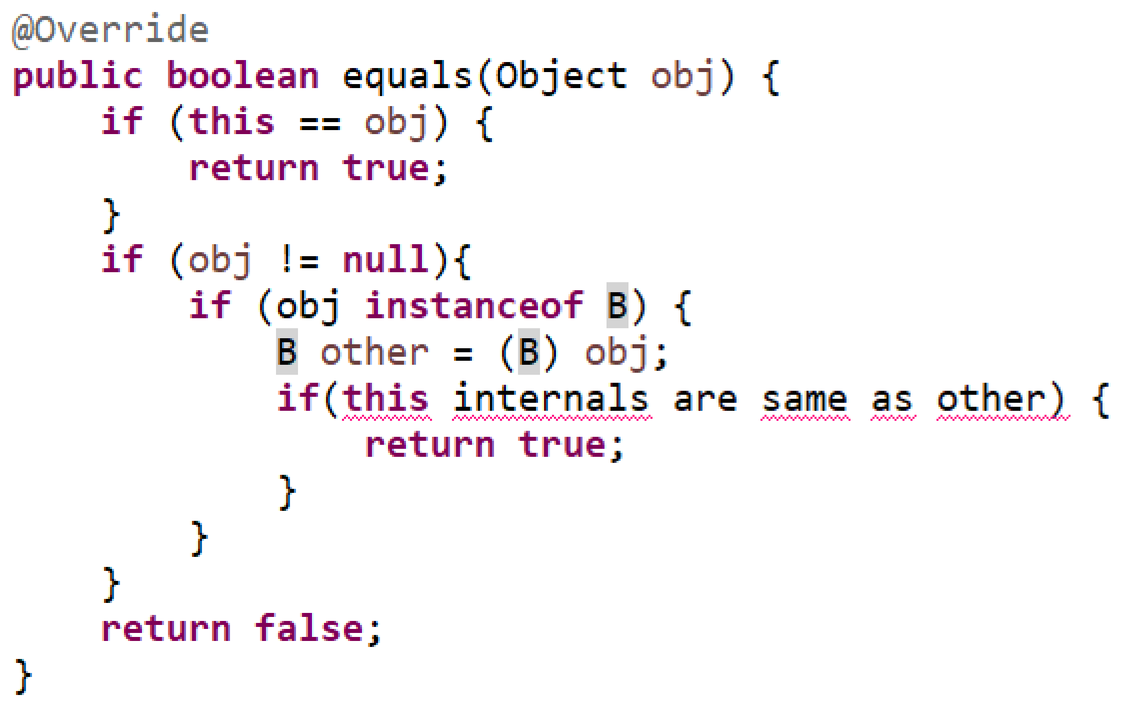
1. Check if obj == this -> not necessary but efficient
2. Check if obj != null -> should always be false if null
3. Check if obj has same class as this class
4. Cast Object to our class type
5. Check if internals of object are the same

public boolean equals(Object obj)

Two examples of class B equals



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



```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj != null){
        if (obj instanceof B) {
            B other = (B) obj;
            if(this internals are same as other) {
                return true;
            }
        }
    }
    return false;
}
```

public boolean equals(Object obj)

There is a debate of

obj instanceof B

Vs

getClass() == obj.getClass()

Strong exact class comparison, or should inheritance be allowed? Usually it doesn't matter, but since eclipse/Netbeans/etc. auto-generate it is considered important what they choose to do.

public boolean equals(Object obj)

```java
public class Person {
    private int id;
    private String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

This Person class considers to people the same if their id is the same, regardless of their name

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

public boolean equals(Object obj)

This Person class considers to people the same if their id is the same, regardless of their name

```java
Person p1 = new Person(1, "James");
Person p2 = new Person(1, "Joe");
Person p3 = new Person(2, "James");
System.out.println(p1 == p2);
System.out.println(p1 == p3);
System.out.println(p1.equals(p2));
System.out.println(p1.equals(p3));
```

```
false
false
true
false
```

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

Onward to … Interfaces

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY