Describe `hashCode()` contract.

If two objects are `equal()` they must have the same `hashCode()`.

When is there no need to create `canEqual` method?

If the class is final and inherits equals from `Any`. There is no issue of subtype equality in this case.

Why not force `equals()` symmetry by checking the equality of runtime classes?

Anonymous class instances won't be equal to non-anonymous ones.

In general, subtypes won't be able to equal supertypes.

Other than Java interop, what is a common use of existential types?

Ignoring type parameters with `MyClass[_]` notation.

e.g., `Map[_,_]`

Give the `hashCode()` recipe.

```
41* (
  41* (
    41* (
      41 + a.hashCode
    ) + b.hashCode
  ) + c.hashCode
) + d.hashCode
```

What is your `hashCode()` only good as?

Since ___ is not great by default, what should you use?

It's only as good as the hash codes you make it out of.

For example, most collections override `hashCode` for you, but Arrays do not.

For `Arrays`, has each element or use `java.util.Arrays.hashCode`.

Why is 41 used in `hashCode` recipe?

Mult: odd primes minimize the potential for information loss on overflow.

Add: avoid the first field being zero, assuming zero is more likely than -41. Any non-zero integer is equally good.

What are the equivalence relation `equals()` requirements for non-null objects?

- Reflexive
- Symmetric
- Transitive
- Consistent: provided info used by equals was not modified
- Not `equal()` to null

Give some common `equals()` pitfalls.

- Defining `equals()` with wrong signature.
- Changing `equals()` without changing `hashCode()`.
- Defining `equals()` in terms of mutable fields.
- Failing to define `equals()` as an equivalence relation.

Give the `equals()` recipe.

```
class X extends Y {
  def canEqual(other:Any) : Boolean = {
    other.isInstanceOf[X]}
override def equals(other:Any) : Boolean =
  other match {
    case that: X => {super.equals(that) &&
    (that canEqual this) && fields match }
    case _ => false
if extending AnyRef no super call,
}
```

What should you do if your `hashCode()` invokes `super.hashCode()`?

Start your `hashCode()` with that invocation.

```
41 * (
   super.hashCode()
   ) + a.hashCode()
) + b.hashCode()
```

Describe efficient `hashCode`.

For immutable objects, override `hashCode` with similarly named val.

For mutable objects, use caching.

What is a common source of `equals()` methods that are not equivalence relations?

Subtypes.

Say A extends B.

```
val a = new A ; val b = new B
```

"a equals b" and "b equals a" use different versions of equals!
So just overriding A's is insufficient.

What's an apparent inconsistency with Java's `equals()`?

Scala's takes Any instead of `AnyRef`/`Object`.

It's just a fiction of the compiler; it's the same method.

What is `Any`'s `==` defined as?

```
final def == (that:Any) : Boolean =
  if (null eq this) {null eq that}
  else {this equals that}
```