Casting and Runtime Verification
(mostly an at-home reading)

Casting

We have discussed casting being used in two different ways this semester.

- The first was to “narrow” a data type, instructing Java to convert a primitive even if it would mean some information was lost (such as casting a float into an int).
- The second was to “promote” a reference of an interface type to be a reference to a type that implements that interface. This second use can be useful when dealing with generic data structures such as ArrayList<T>. Note that in Java this casting just alters the reference, not the object itself.

Of course, we need to be sure we are correct or at runtime an exception will be thrown.
Methods not in the interface...

Thinking back to our Animal interface and our Cat and/or Dog classes, an Animal reference pointing to a Cat or Dog object cannot call a method that isn’t specified in the Animal interface which was why if a program had

```
Animal pet = new Dog("Fluffy");
```

we would not be able to then say

```
pet.buryBone();
```

even though a Dog object can invoke that method, because pet isn’t known by the compiler to be a reference to a Dog, just a reference to an Animal.

---

Casting

If our program had Animal pet = new Dog("Fluffy"); we would not be able to then say pet.buryBone(); but we could say

```
((Dog)pet).buryBone();
```

and the compiler would “trust” us because it knows Dog implements Animal, so it could be valid.

However, at runtime Java will “verify” our claim, and if it detects the type of the object is not what you said it would be, an exception will be thrown.
ClassCastException

If our program had

```
Animal pet = new Cat("Crookshanks");
```

and we wrote

```
((Dog)pet).buryBone();
```

the compiler would “trust” us because it knows `Dog` implements `Animal`, but at runtime when Java goes to “verify” our claim and detects the type of the object is a `Cat` rather than a `Dog`, a `ClassCastException` exception will be thrown.

---

The `equals` method.

In Java, for reasons that you will learn about in CMSC132, the `equals` method of a class should have the signature

```
public boolean equals(Object other)
```

but within the method, the object to which `other` refers needs to be accessed with a reference of the data type of the current class.
public boolean equals(Object other) {
    try {
        Cat localCat = (Cat)other;
        return getName().equals(localCat.getName());
    }
    catch (Exception e) {
        return false;
    }
}