The instanceof keyword

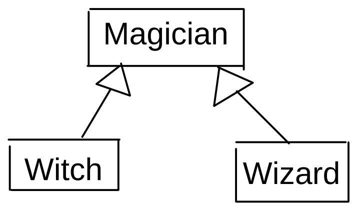
To test if an object is a specific data type, we can use the instance of keyword.

Exampe:
```
   NinjaCat c = new NinjaCat("DonkeyCello");

   if (c instanceof NinjaCat) // condition is true
   {

   }

   if (c instanceof Animal) // condition is true
   {

   }

   if (c instanceof PirateDog) // condition is false (assume we have a class called PirateDog that extends Animal)
   {

   }
```

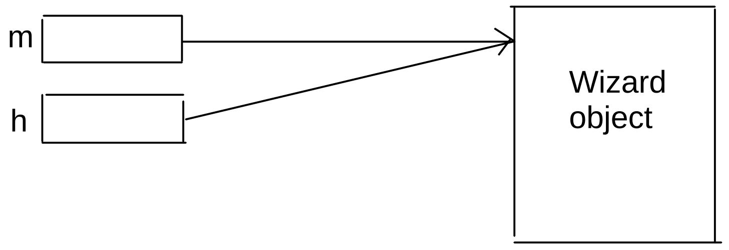
Casting objects

Casting is a specific type of operator that converts a value or reference from one type to another.

Suppose we have the following class hierarchy:



```
Magician m = new Wizard("Harry Potter");
// ...
Wizard h = (Wizard)m; // This casts m to a Wizard. Put type we are converting to in parentheses
                      // Note: m's type doesn't change. The cast returns a reference of the new type
```



Casting an object reference from a supertype to a subtype is called a downcast.
Casting an object reference from a subtype to a supertype is called an upcast.

An upcast is always safe to do. But doing a downcast isn't always safe to do.
So, for a downcast we always have to write the cast. But we do not need to write the cast for an upcast.

```
Witch k = new Witch("Kiki");
Magician n = k; // upcast, so no need for (Magician) before the k

Wizard w = (Wizard)n; // Error: n does not point to a Wizard
```

Interfaces

Allowing a class to (directly) extend more than one class is called multiple inheritance.
There are many complex issues with multiple inheritance. Java avoids these issues by not supporting multiple inheritance. Instead, Java uses single inheritance along with interfaces.

An interface defines some behavior (like comparing objects for equality), and classes that behave in that way will implement the interface.

An interface is like an abstract class except ALL of its methods are abstract and it does not have any non-static data fields. The behavior is specified by declaring methods in the interface. Classes that implement the interface will inherit these methods (and override them if the class is not abstract).

Notes:
- An abstract class can have non-abstract methods, but ALL of the methods in an interface are abstract.
- In an interface, we do not use the keyword abstract in its method declarations.
- All members of an interface are implicitly public (no need to include public in declarations in interfaces).
- All data fields in an interface are static and final. You cannot have a non-static data field in an interface.
- Usually you only include methods in an interface (no data fields).
- Methods in an interface can have default implementations.

Example (put in file Drawable.java)

```
public interface Drawable
{
   void draw(); // No need to declare abstract or public. Java already know this since this is the only option.
}                    // No blody supplied for draw() since it is abstract. End in a ;
```

As with abstract classes, the data type an interface defines is abstract. So we cannot instantiate (make) an instance of an interface.

We cannot do:

```
Drawable d = new Drawable(); // Error, cannot call new Drawable();
```

To make a subtype of an interface, use the keyword implements.

```
public class Image implements Drawable
{
   // ...

   @Override
   public void draw()   // must override since inheritd method draw() is abstract and Image is not abstract
   {
      // ...
   }
}
```

Note: An interface can implement other interfaces. Inherited methods are not implement in subinterfaces.

A non-abstract class must implement (override) ALL non-static methods inherited from its interfaces (unless the method has a default implementation).

In Java, a class can only extend on superclass, but it can implement many interfaces.

```
public class C extends B implements A1, A2, A3
{
   // C inherits members from B, A1, A2, and A3
}
```

In some method that uses C:

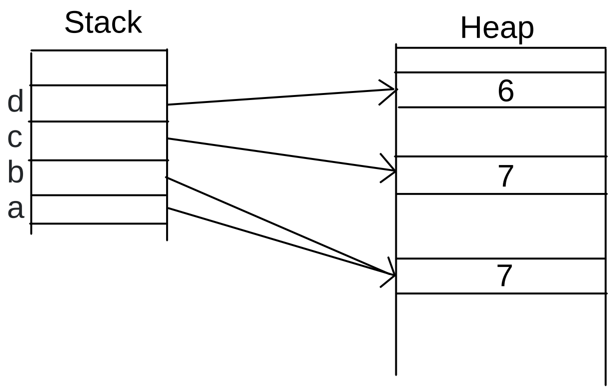
```
   A1 x = new C(); // A variable's type can be an interface, but the object it references must be a more specific
                   // subtype of that interface
```

Comparing objects

To test if two references "point" to the same object in memory, we use the == operator.
This is called a shallow comparison (for equality).

In some method:

```
Integer a = 7;
Integer b = a;
Integer c = 7;
Integer d = 6;
// Suppose Java doesn't optimize and
// point c to a's object
```



What does a == b return? Ans: true
What does a == c return? Ans: false
What does a == d return? Ans: false

Comparing the actual values of objects (and not where the objects live in memory) is called a deep comparison (for equality).

Class Object contains the method

```
public boolean equals(Object obj)
```

The purpose of this method is to specify what is means for two non-null objects to be equal.

What does a.equals(b) return? Ans: true
What does a.equals(c) return? Ans: true
What does a.equals(d) return? Ans: false

Class Object's version of equals does the following:
x.equals(y) returns true if x == y and it returns false otherwise.
If we do not override equals in our class, then this is what it will do. To have equals do a deep comparison, we must override it do to the comparison.

```java
public class Boat
{
    private int length;
    private int maxOccupancy;

    public Boat(int length, int maxOccupancy)
    {
        this.length = length;
        this.maxOccupancy = maxOccupancy;
    }

    // Follow these steps when overriding equals to do a deep comparison
    @Override
    public boolean equals(Object obj) // Parameter type must be Object
    {
        // Step 1: Test if obj is a Boat
        //         If not, then return false
        if (!(obj instanceof Boat)) return false;

        // Step 2: Cast obj to a Boat
        Boat rhs = (Boat)obj; // rhs stands for Right Hand Side

        // Step 3: Test if the data fields of the invoking object are equal to the ones in rhs
        //         using a deep comparison and return this result. Use == for primitive types,
        //         use .equals for objects.
        //         If the superclass overrides its version of equals to do a deep comparison,
        //         then insert
        //            super.equals(obj) &&
        //         just after return and before length == rhs.length &&
        return length == rhs.length &&
               maxOccupancy == rhs.maxOccupancy;
    }
// ... (More to add later)
```

Whenever you override method equals you must also override method hashCode()
hashCode() is used by collections such as HashMap and HashSet. Failing to override hashCode() in a class that overrides equals will cause these collections to not work properly with your class.

hashCode() returns an int that approximately "uniquely" identifies the object.

There are many different ways hashCode can be overridden.
We will show one way of doing it, which is taken from "Effective Java" (3rd edition) by Josh Bloch.

Add to Boat class:

```
@Override
public int hashCode()
{
    int result = Integer.hashCode(length);
    result = 31 * result + Integer.hashCode(maxOccupancy);
    return result;
}
```

Steps:
1) In hashCode() create an int variable called result, and initialize this to the hash code c for the 1st significant field in your object as computed in step 2a. A significan field is a field that affects equals comparison.
2) For each remaining significant field f do:
   a) Compute an int hash code c for f:
      i) If the field is a primitive, compute Type.hashCode(f) where Type is the wrapper class for f's type.
      ii) If the field is an object and the equals method (of the class containing f) recursively calls equls to compare f, then recursively call hashCode on f. If a more complex comparison is required, compute a "cononical representation" for f and invoke hashCode on it. If f is null, then use 0.
      iii) If the field is an array, then treat it as if each significant element were a separate field.
      If the array has no significant element, use a non-zero constant.
      If all elements in the array are significant, then use Array.hashCode(f).
   b) Combine the hash code c computed in step 2a into result as follows:
      result = 31 * result + c;
3) return result.