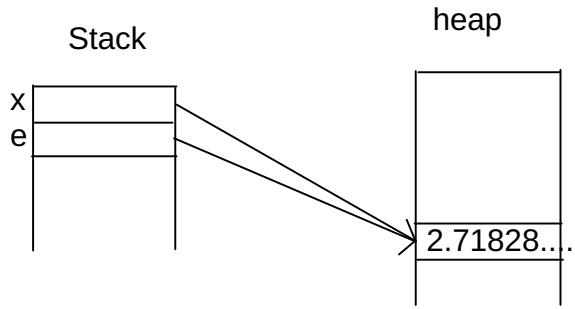
Copying objects

A shallow copy only copies the reference to an object in memory.
Shallow copying of reference types is done with = in Java.

Example:
```
Double e = 2.718281828459;
Double x = e;
```



A deep copy makes a copy of the actual object being referenced, creating a new object in memory whose value is equal to the original (should be equal using the equals method for comparison).

One way to make a deep copy in Java is to use the clone() method. Many programmers do not like to use clone() because it is overly complex and is inconsistent with regular use of interfaces. Regardless, it is still the standard way of making deep copies in Java.

The clone() method is defined in class Object and is protected.
```
protected Object clone() throws CloneNotSupportedException
```

The throws CloneNotSupportedException part indicates that an exception (abnormal situation) might occur if cloning objects is not supported by your class and we try to use clone on it (more on the topic of Exceptions later in the course).

To have clone() do a deep copy of an object of a class, the class must
1) implement the Cloneable interface
2) properly override the clone() method to make a deep copy

The only purpose of interface Cloneable is to mark the fact that an instance of a class can be cloned use method clone(). Cloneable is an example of a marker interface (used to mark some property that a type will have).

Cloneable is defined in java.lang as follows:

```
public interface Cloneable
{
}
```

Example: Modifying class Boat to allow deep copying with clone()
Add to Boat class:

```
@Override
public Object clone()  // Node: method clone() is redeclared public (this allows the class user to call it)
{
    // Step 1: Add a try-catch block to catch (handle) a CloneNotSupportedException (return null in its catch clause).
    //         This step is not needed if the superclass overrides clone() to not throw a CloneNotSupportedException.
    //         try-catch blocks will be explained in full when we talk about Exceptions in this course.
    try
    {
        // Step 2: Call super.clone() and cast its returned object to a Boat
        Boat objClone = (Boat)super.clone();

        // Step 3: deep copy the data fields from the invoking object into objClone
        objClone.length = length;
        objClone.maxOccupancy = maxOccupancy;

        // Step 4: return objClone
        return objClone;
    }
    catch (CloneNotSupportedException e) // Part of Step 1
    {
        return null;
    }
}
```

Disadvantages of clone() method:
- It is complex.
- It doesn't follow regular use of interfaces.
- We don't have any control over object construction since no constructor is called.
- We cannot manipulate final data fields in clone() because final data fields can only be changed in constructors.

Alternative to clone

This alternative to clone() has two steps:
1) Create a copy constructor (more on this later)
2) Create or override a wapper method around the copy constructor (more on this later)
   Calling this method allows the class user to use polymorphism when making deep copies.
   This step is only needed if you want to allow polymorphism when making deep copies.

Step 1:
A copy constructor of a class C is a constructor that takes exactly one argument in whose type is C and it performs a deep copy of this argument into the invoking object.

Add to Boat class:
```
// Copy constructor
public Boat(Boat other)
{
    length = other.length;
    maxOccupancy = other.maxOccupancy;
}
```

Step 2: (only do if the class is not abstract and you want to allow polymorphism with copying instances of the class)
Create or override a wapper method for the copy constructor. This method takes no arguments and calls the copy constructor to return a deep copy of the invoking object.

Add to Boat class:
```
public Boat copy()
{
    return new Boat(this); // calls copy constructor
}
```

We can now make deep copies of Boat object by calling copy():

```
Boat b1 = new Boat(18, 10);
Boat b4 = new Boat(b1);
Boat b5 = b1.copy();
```