'Or else it doesn't, you know. The name of the song is called "HADDOCKS' EYES."'

`Oh, that's the name of the song, is it?' Alice said, trying to feel interested.
`No, you don't understand,' the Knight said, looking a little vexed. `That's what the name is CALLED. The name really IS "THE AGED AGED MAN."
`Then I ought to have said "That's what the SONG is called"?' Alice corrected herself.
`No, you oughtn't: that's quite another thing! The SONG is called "WAYS AND MEANS": but that's only what it's CALLED, you know!'
`Well, what IS the song, then?' said Alice, who was by this time completely bewildered.
`I was coming to that,' the Knight said. `The song really IS "A-SITTING ON A GATE": and the tune's my own invention.'

- Lewis Carroll, *Through the Looking Glass*
Object Variables

- Object variables are declared by stating the class name / data type and then the variable name
  - same as primitives
  - in Java there are hundreds of built-in classes.
  - don't learn the classes, learn how to read and use a class interface (the user manual)

- Objects are *complex* variables. They have an internal *state* and various *behaviors* that can either change the state or simply tell something about the object

```java
public void objectVariables() {
    Rectangle rect1;
    Rectangle rect2;
    // 2 Rectangle objects exist??
    // more code to follow
}
```
Onto Object Variables

public void objectVariables() {
    Rectangle rect1;
    Rectangle rect2;
    // 2 Rectangle objects exist??
    // more code to follow
}

- So now there are 2 Rectangle objects right?
- Not so much.

- Object variables in Java are actually *references* to objects, not the objects themselves!
  - object variables store the memory address of an object of the proper type *not* an object of the proper type.
  - contrast this with primitive variables
The Pointer Sidetrack

- A pointer is a variable that stores the memory address of where another variable is stored.

- In some languages you can have static variables (nothing to do with the static keyword) and dynamic variables of any type.

- Example C++, can have a, integer variable or a integer pointer (which is still a variable).

```cpp
int intVar; // a int var
int * intPtr; // pointer to an int var
```
```cpp
int intVar = 5; // a int var
int * intPtr; // pointer to an int var
intPtr = new int;
/* dynamically allocate an space to store an int. intPtr holds the memory address of this space*/
```

**intVar**

5

**intPtr**

0x00122155

**Space for an int in memory assume memory address 0x00122155**
Pointer Complications

- C++ allows actual variables and pointers to variables of any type. Things get complicated and confusing very quickly.

```cpp
int intVar = 5; // a int var
int * intPtr;  // pointer to an int var
intPtr = new int;  // allocate memory
*intPtr = 12;  // assign the integer being pointed to the value of 12. Must dereference the pointer. i.e. get to the thing being pointed at*/
cout << intPtr << "\t" << *intPtr << "\t" << &intPtr << endl;
// 3 different ways of manipulating intPtr
```

- In C++ you can work directly with the memory address stored in intPtr
  - increment it, assign it other memory addresses, pointer “arithmetic”
Benefit of Pointers

- Why have pointers?
- To allow the sharing of a variable
  - If several variables (objects, records, structs) need access to another single variable two alternatives
    1. keep multiple copies of variable.
    2. share the data with each variable keeping a reference to the needed data.

![Diagram showing how pointers allow sharing of a variable](image_url)
More Benefits

- Allow dynamic allocation of memory
  - get it only when needed (*stack* memory and *heap* memory)
- Allow linked data structures such as *linked lists* and *binary trees*
  - incredibly useful for certain types of problems
- Pointers are in fact necessary in a language like *Java* where *polymorphism* is so prevalent (more on this later)
- Now the good news
  - In *Java* most of the complications and difficulties inherent with dealing with pointers are removed by some simplifications in the language
Dynamic Memory Allocation

Your program has two chunks of memory to work with: *Stack memory* (or the *runtime Stack*) and *Heap memory*

When a Java program starts it receives two chunks of memory one for the Stack and one for the Heap.

Things that use Stack memory: local variables, parameters, and information about methods that are in progress.

Things that use Heap memory: everything that is allocated using the `new` operator.
void toyCodeForMemory(int x) {
    int y = 10;
    x += y;
    String s = new String("Hello");
    System.out.println(x + " " + y + s);
}
How Much Memory?

How big is the Heap?
System.out.println("Heap size is " +
Runtime.getRuntime().totalMemory());

How much of the Heap is available?
System.out.println("Available memory: " +
Runtime.getRuntime().freeMemory());
Pointers in Java

- In Java all primitive variables are value variables. (real, actual, direct?)
  - it is impossible to have an integer pointer or a pointer to any variable of one of the primitive data types

- All object variables are actually *reference variables* (pointers, memory addresses) to objects.
  - it is impossible to have anything but pointers to objects. You can never have a plain object variable
Back to the Rectangle Objects

- rect1 and rect2 are variables that store the memory addresses of Rectangle objects.
- right now they are uninitialized and since they are local, variables may not be used until they are given some value.

```java
public void objectVariables() {
    Rectangle rect1;
    Rectangle rect2;
    // rect1 = 0;       // syntax error, C++ style
    // rect1 = rect2;   // syntax error, uninitialized
    rect1 = null;     // pointing at nothing
    rect2 = null;     // pointing at nothing
}
```

- null is used to indicate an object variable is not pointing / naming / referring to any Rectangle object.
Creating Objects

- Declaring object variables does *not* create objects.
  - It merely sets aside space to hold the memory address of an object.
  - The object must be created by using the `new` operator and calling a constructor for that object.

```java
public void objectVariables() {
    Rectangle rect1;
    rect1 = new Rectangle();
    Rectangle rect2 = new Rectangle(5, 10, 20, 30);
    // (x, y, width, height)
    // rect1 and rect2 now refer to Rectangle objects
}
```

- For all objects, the memory needed to store the objects, is allocated dynamically using the `new` operator and a constructor call. (Strings are a special case.)
  - constructors are similar to methods, but they are used to initialize objects.
The Yellow Sticky Analogy

- **rect1**
  - Rectangle Object:
    - x: 0
    - y: 0
    - width: 0
    - height: 0

- **rect2**
  - Rectangle Object:
    - x: 5
    - y: 10
    - width: 20
    - height: 30
Pointers in Java

- Is this easier?
  - primitives one thing, objects another?

- can't get at the memory address the pointer stores as in C++
  although try this:
  ```java
  Object obj = new Object();
  System.out.println( obj.toString() );
  ```

- dereferencing occurs automatically

- because of the consistency the distinction between an object and an object reference can be blurred
  - "pass an object to the method" versus "pass an object reference to the method"

- Need to be clear when dealing with memory address of object and when dealing with the object itself
The `toString()` method is useful for debugging. By default, when an object is printed out in a print stream like `System.out`, the `toString()` method of the object is automatically called. For example, in the following code, the string representation of `myObject` is printed out:

```java
MyClass myObject = new MyClass();
System.out.println("myObject=
" + myObject);
```

The problem with just printing out `myObject` is that the default `toString()` method of the superclass (Object) is called unless `myObject` overrides the `toString()` method. As a result, if `myObject` does not override the `toString()` method, just the reference value of `myObject` (not its contents) is printed out. For example, the previous call may print out something like:

```
myObject = MyClass@1ee80a
```

What you'd like to see is a printout of the contents of the object, not the object's reference.

Vagelis Hristidis; 11.01.2006
Working with Objects

- Once an object is created and an object variable points to it then Object may be manipulated via its methods

```java
Rectangle r1 = new Rectangle();
r1.resize(100, 200);
r1.setLocation(10, 20);
int area = r1.getWidth() * r1.getHeight();
Rectangle r2 = null;
r2.resize(r1.getWidth(), r1.getHeight() * 2); // uh-oh!
```

- Use the dot operator to deference an object variable and *invoke* one of the objects behaviors

- Available behaviors are spelled out in the class of the object, (the data type of the object)
What's the Output?
(Or, do you understand how object variables and pointers work?)

```java
public void objectVariables(String[] args) {
    Rectangle rect1 = new Rectangle(5, 10, 15, 20);
    Rectangle rect2 = new Rectangle(5, 10, 15, 20);
    System.out.println("rect 1: " + rect1.toString()
    );
    System.out.println("rect 2: " + rect2.toString()
    );
    System.out.println("rect1 == rect2: " + (rect1 == rect2));
    rect1 = rect2;
    rect2.setSize(50, 100); // (newWidth, newHeight)
    System.out.println("rect 1: " + rect1.toString()
    );
    System.out.println("rect 2: " + rect2.toString()
    );
    System.out.println("rect1 == rect2: " + (rect1 == rect2));
    int x = 12;
    int y = 12;
    System.out.println("x == y: " + (x == y)
    );
    x = 5;
    y = x;
    x = 10;
    System.out.println("x == y: " + (x == y)
    );
    System.out.println("x value: " + x + "y value: " + y);
}
```
Equality versus Identity

A man walks into a pizza parlor, sits down, and tells the waiter, "I'll have what that lady over there is eating." The waiter walks over to the indicated lady, picks up the pizza that is resting in front of her, and sets it back down in from of the man's table.

- confusion over equality and identity
- identity: two things are in fact the same thing
- equality: two things are for all practical purposes alike, but not the exact same thing
- == versus the .equals method
  - use the .equals method when you want to check the contents of the pointee, use == when you want to check memory addresses
The Garbage Collector

Rectangle rect1 = new Rectangle(2, 4, 10, 10);
Rectangle rect2 = new Rectangle(5, 10, 20, 30);
// (x, y, width, height)
rect1 = rect2;
/* what happened to the Rectangle Object
   rect1 was pointing at?
*/

- If objects are allocated dynamically with new how are they deallocated?
  - delete in C++

- If an object becomes isolated (no longer is in scope), that is has no references to it, it is garbage and the Java Virtual Machine garbage collector will reclaim this memory AUTOMATICALLY!
Objects as Parameters

- All parameters in Java are *value* parameters
- The method receives a copy of the parameter, not the actual variable passed
- Makes it impossible to change a primitive parameter
- implications for objects? (which are references)
  - behavior that is similar to a reference parameter, with a few minor, but crucial differences
  - "Reference parameter like behavior for the pointee."
Immutable Objects

- Some classes create *immutable* objects
- Once created these objects cannot be changed
  - note the difference between objects and object variables
- Most immediate example is the String class
- String objects are immutable
- Why might this be useful?

```java
String name = "Mike";
String sameName = name;
name += " " + "David" + " " + "Scott";
System.out.println( name );
System.out.println( sameName );
```
public void setupHash()
{
    Hashtable ht = new Hashtable();
    String key = "key1";
    ht.put(key, new Vector());
    key = "key2";
    Vector vec = (Vector)ht.get("key1");
    vec.addElement(new Integer(5));
}

In the third line of this method, the object key is added to the hash with a new object. At this point, the hash will contain a String object for the key and an instantiated Vector object as its value. In the fourth line, key is given a new value, "key2". Or did it? In fact, since key is instantiated from the immutable String class, key is now a separate object from the one that was put into the Hashtable, and two String objects exist in memory. In the fifth line, the value "key1" (the third String created) is used to look up the Vector in the Hashtable and get a reference to it, and in the sixth line an object is successfully added to it.

Vagelis Hristidis; 11.01.2006