Objects: Where?  Heap store

Objects are more permanent
- method called
  - method creates object
  - method returns
    - object lives on

... can't store objects in frames
The memory where objects are stored
  is called the heap.
public class Date{
    private int day;
    private String month;
    private int year;
}

- instance variables (non-static fields) - stored in each instance (object) of class Date

Each instance of Date laid out as -

reference → a String object

object header day month year
Memory contains:

- frames - seen already
- heap
  objects (including arrays)
- method area
  method bytecode
  constant pools
  class variables (static fields)

chain of linked frames is called a stack, don't confuse it with operand stacks
Stack
one record (frame) per method call
parameters, local variables
operand stacks

Heap
one record per object (instance)
instance variables (non-static fields)

Method area
one record per class
constant data: constant pool, method bytecode
class variables (static fields)

[Red text]: you’ll see these later in SW workshop
An object

Number of references to this object *** explained later

Object header includes these

Address of method area for class of this object

<table>
<thead>
<tr>
<th>ref count</th>
<th>class</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
For local variables:
use iload, fload etc.

Accessing instance variables using bytecode

getfield 2-byte index

..., object ref ⇒ ..., value of variable
pushes variable value onto operand stack
index is index into constant pool
- object reference is address of object
- constant pool entry is symbolic reference -
  names of instance variable + class declaring it
In the stack frame, the program counter (PC) is set to the top of the stack. The stack frame contains a reference to an object. On the heap, a value is pushed onto the operand stack. This value represents an object, which is stored in memory. The class representing the method being executed and the class where the field is declared are also shown. The field layout for instances of this class is indicated.
Accessing fields using bytecode

`putfield 2-byte index`

is similar to pop value from operand stack:

```null
..., object ref, new field value => ...
```
Calling an instance method

invokevirtual 2-byte index

Similar to invokevirtual for static methods but operand stack has ref to "this" object that will execute method

Method calls - static
invokestatic 2-byte index
- index is used as index into constant pool of current class (of method currently being executed)
- constant pool entry must be for a method from it can be found
  - number & types of parameters
  - result
  - address of bytecode for method
  - size & operand stack & needed number & local variables

access permission also checked

--- objectref, actual params => ---

copied to variable #0 in frame
Payable x = new Invoice(29, 6, "printer cartridge", "222333", 31);
int a = x.paymentAmount();
Rule - of "method dispatch"

It is always the class of the object that governs which method definition is chosen.

- This makes a difference if the type of the variable is itself a class.
- It means the method must be chosen at run-time - the compiler can't work it out.
- C++ is different! But the Java rule is secure.
Which method definition?
E.g. to String

- If you don't define to String:
  There's a default definition (in Object class)
  Your class inherits the default

- If you do:
  Your definition overrides the default
Chains of overriding

- arise from class extension (subclasses)

invoke virtual must find correct definition (that overrides all the others)

Method must be defined somewhere - otherwise class is abstract, so can't have instances. (Compiler enforces this.)
Creating an object in bytecode

Two steps

1. Allocate space, with default values
2. Execute constructor to do proper initialization
Creating an object in bytecode

1. Allocate space
   new 2-byte index

- index is to constant pool entry with name of class for new object
- finds class definition
- allocates heap space for object with correct fields, initialized to defaults
- pushes reference to new object

    ....  \implies  ...., object ref
Creating an object in bytecode

2. Execute constructor
   invokespecial  2-byte index
   - index into constant pool gives entry for constructor
   - expects ref to new object on stack
     ...
   , objectref, parameters ⇒ ...

1.2 usually done as
   new ...
   dup    duplicates top of stack
   invokespecial so objectref still on stack at end

cf. invokevirtual
Date myBirthday = new Date(36, "Julember", 2015);

```java
0: new           // class introscs/Date
3: dup
4: bipush 36
6: ldc #14 // String Julember
8: sipush 2015
11: invokespecial #15 // Method
   "<init>" (ILjava/lang/String;I)V
14: astore_1
```

= constructor

int string int three parameters
Date myBirthday = new Date(36, "Julember", 2015);
Date myBirthday = new Date(36, "Julember", 2015);

Example:

```
0: new           #13    // class introscs/Date
3: dup
4: bipush 36
6: ldc #14    // String Julember
8: sipush 2015
11: invokespecial #15    // Method
   "<init>" (ILjava/lang/String;I)V
14: astore_1
```
Verifier checks on object creation

1. new always followed by appropriate invokespecial
2. Extra checks associated with class extension

⇒ object always created consistent with what its designers intend
Some parts used for objects, some parts free

When new object is created: space found in free areas.
When object is destroyed: its space made free again.
Garbage collection

In some languages (e.g. C++) you destroy objects explicitly when you have finished with them. Then their storage area can be recycled. In Java you don't. The garbage collector is part of the JVM. It works out when objects are finished with and then recycles storage area—marks it "free".
When is an object finished with?
Each object is like Isla de Muerta in Pirates of the Caribbean:
"It's an island that cannot be found, except by those who already know where it is."
To know where an object is:
- have a reference to it.
When there are no references to an object:
- it cannot be used any more.
- garbage collector can safely delete it.
objects in heap

reference counts in red

local variables in frames

1

2

0

1

2

1

1

1

1
Reference counts in red

Objects in heap

Can delete

Local variables in frames
objects in heap

reference counts in red

local variables in frames

now suppose this local variable finishes

2

1

1

2

1
objects in heap:
1
2
local variables in frames:
X0
2
1
reference counts in red:
3