Agenda

- Class exploring
  - Class object, default / common behaviors
- Objects equality
  - Object
  - Collections
- Copying
  - Shallow copy
  - Deep copy
- Squeak object model
- The new method
Part I – Object class, Behaviors, Equality, Copying
Object class

- Single root to inheritance hierarchy.
- Most classes inherit (directly or indirectly) from `Object`
- Inherits from `ProtoObject`
  - What is the super class of `ProtoObject`?
- `Object` instances have:
  - no state (no instance variables)
  - approximately 400 methods.
Methods in `Object` may:

- Define common behavior
  - Not meant to be overridden in derived class.
  - Same functionality for all derived classes.

- Define default behavior
  - Meant to be overridden by derived classes.
Default and Common behavior

- **printString** method defines common behavior:
  
  ```Smalltalk
  printString
  ^self printStringLimitedTo: 50000
  ```

- **printStringLimitedTo**: calls **printOn** method.

- **printOn** method defines default behavior:
  
  ```Smalltalk
  printOn: aStream
  | title |
  title := self class name.
  aStream
  nextPutAll: (title first isVowel ifTrue: ['an '] ifFalse: ['a ']);
  nextPutAll: title
  ```
Example – Fraction class

- Class Browser: (method code)

```plaintext
printOn: aStream
    aStream nextPut: $(.
    numerator printOn: aStream.
    aStream nextPut: $/.
    denominator printOn: aStream.
    aStream nextPut: $).
```

- Workspace:

```
Transcript show: 2/5 printString
```

- Transcript:

```
(2/5)
```
Equality Tests

- The method `==` is implemented in `ProtoObject`, by the usage of `<primitive>` direction.
  - It compares objects by reference.
  - Supplies common behavior
- `~~` is based on `==` (also common behavior):

  ```
  ~~ anObject
    anObject == self
    ifTrue:[^ false]
    IfFalse:[^ true]
  ```

- `=` is first defined in class `Object`
  - Supplies default behavior
    ```
    = anObject
      ^self == anObject
    ```
**Collections Equality**

- **SequenceableCollection** overrides '==' behavior, more suitable for its class.

```plaintext
= t1
  self == t1 ifTrue: [^ true].
  self species == t1 species ifFalse: [^ false].
  ^ self hasEqualElements: t1
```

- Thus supplying a common behavior to all derived classes.
  - Depends on **hasEqualElements**, which uses the method **at:**, which defines default behavior.
The method `shallowCopy` returns a copy of the receiver.

- shallow copy shares references with original object.
- `shallowCopy` provides common behavior, based on `basicSize` and `basicAt`.
  - Why must these two methods be primitive?
- `copyTwoLevel` makes a shallow copy for each member.
- `deepCopy` copies members recursively (is termination guaranteed?).
Method **copy** can either be a shallow copy or a deep one:

```plaintext
^ self shallowCopy postCopy
```

By overriding **postCopy** one can make copies of members that must not be shared.

```plaintext
postCopy
| aLink |
super postCopy.
firstLink isNil ifFalse: [
aLink := (firstLink := firstLink copy).
[aLink nextLink isNil] whileFalse:
    [aLink nextLink: (aLink := aLink nextLink copy)].
lastLink := aLink].
```
Part II – Squeak Object Model
Rule I: *Everything* is an Object

- An object consists of:
  - **Identity** (usually the address)
  - **State** (i.e., the value of its members)
    - Fields access modifier is `object protected`
  - **Behavior** (Method implementation)
    - Methods are public (can only be private by convention).
- All objects (except `SmallInteger`, `ByteSymbol`) have reference semantics.
- Variables are dynamically typed.
• A class specifies the structure and behavior of its instances.
  • All instances of a class have the same behavior.
  • Instances of the same class may vary in state.
• Classes are also objects...
• Obtaining the class of an object:

  1 class → SmallInteger
  20 factorial class → LargePositiveInteger
  '236703' class → String
  (1-> 'Parker') class → Association
  {1. 1. 3} class → Array
Rule III – Every class has a single superclass

- Squeak has single inheritance.
- Each class inherits its structure and behavior from another class.

<table>
<thead>
<tr>
<th>Class</th>
<th>Superclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Integer</td>
</tr>
<tr>
<td>Integer</td>
<td>Number</td>
</tr>
<tr>
<td>Number</td>
<td>Magnitude</td>
</tr>
<tr>
<td>Magnitude</td>
<td>Object</td>
</tr>
<tr>
<td>Object</td>
<td>ProtoObject</td>
</tr>
<tr>
<td>ProtoObject</td>
<td>nil</td>
</tr>
</tbody>
</table>
Rule IV – All actions are produced by passing messages

- Everything happens by sending messages to objects
- Except:
  - Atomic commands
  - Native methods
The method to be invoked is searched for in the *class* of the message receiver.

If the method is not found, the method search continues in the superclass.

In case a method to handle a message could not be found, the receiver sends:

```small
self doesNotUnderstand: <message name>
```
Method Lookup

Calling aB foo:

c := self class.

1. Look up the method in the method dictionary of c.
2. If not found then
c := c superclass.
3. If c is nil then ???
   else goto 1.
Reflection

- **respondsTo**: in class **Object**:

  ```lisp
  respondsTo: aSymbol
  ^ self class canUnderstand: aSymbol
  ```

- **canUnderstand**: defined in **Behavior**:

  ```lisp
  canUnderstand: t1
  (self includesSelector: t1) ifTrue: [^ true].
  superclass ifNil: [^ false].
  ^ superclass canUnderstand: t1
  ```

- **includeSelector**: defined in **Behavior**:

  ```lisp
  includesSelector: aSymbol
  "Answer whether the message whose selector is the argument
  is in the method dictionary of the receiver's class."
  ^ self methodDict includesKey: aSymbol
  ```
Since Smalltalk is *dynamically typed* it is useful to ask about the identity of the objects.

The methods `isMemberOf:` and `isKindOf:` defined in `Object` may be used for type checking:

```smalltalk
isMemberOf: t1
  ^self class == t1

isKindOf: t1
  self isMemberOf: t1
  ifTrue: [^ true].
  ^ self class inheritsFrom: t1
```
• Reminder:
  • Everything is an object
  • Every object is an instance of a class

• Thus, logically:
  • Every class is an object
  • Every class object is an instance of a class.
  • What is this “class of a class?”
    • meta-class
Meta-classes and **Meta-class**

- **Class**: (Examples: `Integer`, `Number`, `Array`)
  - Every object is an instance of a class
  - All classes eventually inherit from `ProtoObject`.
  - A class is an instance of its corresponding meta-class.

- **Meta-class**: (Examples: `Integer class`, `Array class`):
  - Inheritance hierarchy is parallel to class hierarchy.
  - All meta-classes eventually inherit from class `Class`.
    - Which eventually inherits from `ProtoObject`.
  - All meta-classes are instances of `MetaClass`.

- **MetaClass**:
  - Sort of a meta-meta-class
Metaclass

- Every class is an object too!
- Meta-classes have only one instance – the class object (classes are singletons) - Why?
Metaclasses are implicit.
- Created implicitly when classes are created.
- Each class has a unique metaclass.

Metaclasses are anonymous (cannot be directly referred from code).

Integer class → Integer class
Object class → Object class
Meta class inheritance hierarchy

- **Object**
  - **Behavior**
    - **ClassDescription**
      - **Class**
        - **ProtoObject class**
          - **Object class**

- **Instance variables**
- **Method categories**
- **Change set and logging**

- **Super class**
- **Method dictionary**
- **Code compilation**
- **new**

- **Class name**
- **The set of subclasses**
- **Class categories**
- **fileOut**
- **etc.**
Metaclasses and classes

- A bit lighter than Class
- new creates a singleton class

All meta-classes are singleton classes, but Metaclass isn’t!
What we’ve learned so far:

• Every class object is an instance of a class (a meta-class)
• Every meta-class is an object.
• Every meta-class object is an instance of a class (Metaclass)

So, according to the rules:

• Metaclass is an object
• Metaclass is an instance of a class
  • By squeak naming conventions Metaclass class
  • Which is a meta-class... (class of a class)
  • So it’s also an instance of Metaclass....
**Metaclas and its Meta-class**

- **Metaclass** is an instance of **MetaClass class**
- **MetaClass class** is an instance of **Metaclass**

Eventually inherits from **Class**

Inherits from **ClassDescription**
A class has the following attributes:

- Name
- Set of methods
- Set of instance variables
- Instance size

A class has a behavior, defined by the methods in its class (meta-class).

Factory method (new) – creates new instances.

Looked up in the *meta-class* inheritance hierarchy.
Types of Variables

Defined in the class:
• Instance Variables

Defined in the meta-class:
• Class Variables
  – Like static variables in C++/Java
• Class-Instance Variables
  – Instance variables for the class object.
The method **new**

- First defined in class **Behavior**.
- May be overridden in derived meta-classes.
  - **new** always returns an instance of **self**
    (self is a class, can be instantiated)
- New first creates a new instance, and then sends an **initialize** message to it.

```ruby
new
  ^ self basicNew initialize
```

- **basicNew** allocates memory using `<primitive>`.
  - When overriding **new**, make sure **basicNew** is called at some point.
new Method (cont.)

- new is redefined in Metaclass:
  - Meaning the behaviors of its instances change: the meta-classes.

```
new
"The receiver can only have one instance. Create it or complain that one already exists."
  thisClass class ~~ self
  ifTrue: [^ thisClass := self basicNew].
  self error: 'A Metaclass should only have one instance!'
```

- So every instance of Metaclass is in fact a singleton class.
Summary questions

• Why is there a meta-class for every class, but only one `MetaClass`?

• When looking up a method, `methodDict` should be searched. But fields are protected, so a method should be used. That method should be defined in the meta-class, so another lookup is required. How is an infinite recursion avoided as the lookup deepens?