Messages, Instances and Initialization

עדוא מרצィ
המחלקה למדעי המחשב
אוניברסיטת בן-גוריון
Dynamic Aspects of Classes

In the last chapter we described the static, or compile time aspects of classes.

In this chapter we examine their run-time features:

- How values are instantiated (or created) ?
- How they are initialized ?
- How they communicate with each other by means of message passing ?
Roadmap

In this chapter we examine:

- Message passing syntax
- Object creation and initialization (constructors)
- Accessing the receiver from within a method
- Memory management or garbage collection
Messages are not Function Calls

Recall that we noted the following differences between a message and a function (or procedure) call:

- A message is always given to some object, called the receiver
- The action performed in response is determined by the receiver
- Different receivers can do different actions in response to the same message
Message Passing Syntax

Although the syntax may differ in different languages, all messages have three identifiable parts:

1. The message receiver
2. The message selector
3. An optional list of arguments

```java
aGame.displayCard(aCard, 42, 27);
```
Message Passing Syntax

- Smalltalk and Objective-C use a slightly different syntax:

  aGame display: aCard atLocation: 45 and: 56

- In these languages a **space is used as a separator**

- Unary messages (messages that take no argument) are simply written following the receiver

- Messages that take arguments are written using **keyword notation**

- The **message selector** is split into parts, one part before each argument

- A colon follows each part of the key
Message Passing Syntax

- In **Objective-C** a Smalltalk-like message is enclosed in a pair of square brackets.
- Termed a *message passing expression*.
- The brackets *only surround the message itself*.

```c
int cardrank = [ aCard getRank ]
```
Binary Operations

- In **Smalltalk** even binary operations, such as addition, are interpreted as a message
  
  - Sent to the left value with the right value as argument:

  \[ z <- x + y \]  
  "message to \( x \) to add \( y \) to itself and return sum"

- It is possible to define binary operators in **C++** to have similar meanings

- Example: **Vector arithmetic**
## Message Passing in Various Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
</table>
| C++, C#, Java, Python, Ruby | aCard.flip ();  
aCard.setFaceUp(true);  
aGame.displayCard(aCard, 45, 56); |
| Pascal, Delphi, Eiffel, Oberon | aCard.flip;  
aCard.setFaceUp(true);  
aGame.displayCard(aCard, 45, 56); |
| Smalltalk           | aCard flip.  
aCard setFaceUp: true.  
aGame display: aCard atLocation: 45 and: 56. |
| Objective-C         | [ aCard flip ].  
[ aCard setFaceUp: true ].  
[ aGame display: aCard atLocation: 45 and: 56 ] |
| CLOS                | (flip aCard)  
(setFaceUp aCard true)  
(displayCard aGame 45 56) |
Statically Typed and Dynamically Typed Languages

- A **statically typed language** requires the programmer to declare a type for each variable
  - The validity of a message passing expression will be checked at compile time
  - based on the declared type of the receiver
  - Java, C++, C#

- A **dynamically typed language** associates types with values, not with variables
  - A variable is just a name
  - The legality of a message cannot be determined until run-time
  - Smalltalk, Python, CLOS
The Receiver Variable

Inside a method, the receiver can be accessed by means of a pseudo-variable

- Called **this** in Java, C++, C#
- Called **self** in Smalltalk, Objective-C, Object Pascal
- Called **current** in Eiffel

```pascal
function PlayingCard.color : colors;
begin
  if (self.suit = Heart) or (self.suit = Diamond) then
    color := Red
  else
    color := Black;
end
```
Implicit Use of This

Within a method, a message expression or a data access with no explicit receiver is implicitly assumed to refer to this:

```java
class PlayingCard {
    ...
    public void flip () { setFaceUp( !faceUp ); } 
    ...
}
```

Is assumed to be equivalent to:

```java
class PlayingCard {
    ...
    public void flip () { this.setFaceUp( !this.faceUp); } 
    ...
}
```
Explicit use of the pseudo variable

One place where the use of the variable often cannot be avoided is when a method wishes to pass itself as an argument to another function.

As in the following bit of Java:

class QuitButton extends Button implements ActionListener {
  public QuitButton () {
    ...
    // install ourselves as a listener for button events
    addActionListener(this);
  }
  ...
};
Explicit use of the pseudo variable

- Some **style guidelines** for Java suggest the use of `this` when arguments in a constructor are used to initialize a data member.

- **The same name can then be used for the argument** and the data member, with the explicit `this` being used to distinguish the two names:

```java
class PlayingCard {
    public PlayingCard (int suit, int rank) {
        this.rank = rank; // this.rank is the data member
        this.suit = suit;  // rank is the argument value
        this.faceUp = true;
    }
    ...
    private int suit;
    private int rank;
    private boolean faceUp;
}
```
Object Creation

In most programming languages objects can be created **dynamically**, usually using the `new` operator:

```java
PlayingCard aCard; // simply names a new variable

aCard = new PlayingCard(Diamond, 3); // creates the new object
```

- The declaration simply names a variable,
- The new operator is needed to create the new object value
## Syntax used for Object Creation

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td><code>PlayingCard * aCard = new PlayingCard(Diamond, 3);</code></td>
</tr>
<tr>
<td>Java, C#</td>
<td><code>PlayingCard aCard = new PlayingCard(Diamond, 3);</code></td>
</tr>
</tbody>
</table>
| Object Pascal | `var
aCard : ^ PlayingCard;
beginn
  new (aCard);
  ...
end` |
| Objective-C | `aCard = [ PlayingCard new ];` |
| Python   | `aCard = PlayingCard(2, 3)` |
| Ruby     | `aCard = PlayingCard.new` |
| Smalltalk| `aCard <- PlayingCard new.` |
Pointers and Memory Allocation

- All object-oriented languages use pointers in their underlying representation.

- Not all languages expose this representation to the programmer.

- It is sometimes said that “Java has no pointers” as a point of contrast to C++.

- A more accurate statement would be that Java has no pointers that the programmer can see.

- Since all object references are in fact pointers in the internal representation.
Pointers and Memory Allocation

1st reason

- Pointers normally reference memory that is heap allocated.
- And thus does not obey the normal rules associated with variables in conventional imperative languages.
- In an imperative language a value created inside a procedure will exist as long as the procedure is active, and will disappear when the procedure returns.

```java
public static void foo(int a, boolean b) {
    int var_a;
    int var_b;
    ... 
    ... 
}
```
Pointers and Memory Allocation

1st reason

- Pointers normally reference memory that is heap allocated

- And thus does not obey the normal rules associated with variables in conventional imperative languages

- In an imperative language a value created inside a procedure will exist as long as the procedure is active, and will disappear when the procedure returns

- A heap allocated value, on the other hand, will continue to exist as long as there are references to it

- Which often will be much longer than the lifetime of the procedure in which it is created
A second reason is that some languages, notably C++, distinguish between conventional values and pointer values.

In C++, a variable that is declared in the normal fashion, a so-called automatic variable, has a lifetime tied to the function in which it is created.

When the procedure exits, the memory for the variable is recovered:

```c
void exampleProcedure
{
    PlayingCard ace(Diamond, 1);
    ...
    // memory is recovered for ace
    // at end of execution of the procedure
}
```

Values that are assigned to pointers (or as references, which are another form of pointers) are not tied to procedure entry.
Memory Recovery

Because in most languages objects are dynamically allocated, they must be recovered at run-time.

There are two broad approaches to this:

• Force the programmer to explicitly say when a value is no longer being used:

  delete aCard; // C++ example

• Use a garbage collection system that will automatically determine when values are no longer being used, and recover the memory
Memory Errors

Garbage collection systems impose a run-time overhead, but prevent a number of potential memory errors:

1. Running out of memory because the programmer forgot to free values:

2. Using a memory value after it has been recovered:

   ```cpp
   PlayingCard * aCard = new PlayingCard(Spade, 1);
   delete aCard;
   cout << aCard.rank();
   ```

3. Free the same value twice:

   ```cpp
   PlayingCard * aCard = new PlayingCard(Spade, 1);
   delete aCard;
   delete aCard; // delete already deleted value
   ```
How to Manage Memory in an Object Oriented Program?

- When a garbage collection system is not available, to avoid these problems.

- It is often necessary to ensure that every dynamically allocated memory object has a designated owner.

- The owner of the memory is responsible for ensuring that the memory location is used properly and is freed when it is no longer required.

- In large programs, as in real life:

  Disputes over the ownership of shared resources can be a source of difficulty.
How to manage memory in an Object Oriented Program?

- When a single object cannot be designated as the owner of a shared resource, another common technique is to use reference counts:
  - Count of the number of pointers that reference the shared object
  - Care is needed to ensure that the count is accurate
  - When the count reaches zero it indicates that no pointers refer to the object, and its memory can be recovered

- As with the arguments for and against dynamic typing, the arguments for and against garbage collection tend to pit efficiency against flexibility
  - Automatic garbage collection can be expensive, as it necessitates a run-time system to manage memory
  - On the other hand, the cost of memory errors can be equally expensive
A constructor is a function that is implicitly invoked when a new object is created.

The constructor performs whatever actions are necessary in order to initialize the object.

- In C++, Java, C# a constructor is a function with the same name as the class.
- In Python constructors are all named `__init__`.
- In Delphi, Objective-C, constructors have special syntax, but can be named anything.

```java
class PlayingCard { // a Java constructor
    public PlayingCard (int s, int r) {
        suit = s; rank = r; faceUp = true;
    }
    ...
}
```
Linking creation and initialization together

- Linking creation and initialization together has many beneficial consequences.

- Most importantly, it guarantees that an object can never be used before it has been properly initialized.

- When creation and initialization are separated (as they must be in languages that do not have constructors), a programmer can easily forget to call an initialization routine after creating a new value, often with unfortunate consequences.

- A less common problem, although often just as unfortunate, is to invoke an initialization procedure twice on the same object.
Overloaded Constructors

Constructors are often overloaded, meaning there are a number of functions with the same name.

They are differentiated by the type signature, and the arguments used in the function call or declaration:

class PlayingCard {
public:

    PlayingCard ( ) // default constructor,  
                   // used when no arguments are given
        { suit = Diamond; rank = 1; faceUp = true; } 

    PlayingCard (Suit is) // constructor with one argument
        { suit = is; rank = 1; faceUp = true; } 

    PlayingCard (Suit is, int ir) // constructor with two arguments
        { suit = is; rank = ir; faceUp = true; } 
};
The Orthodox Canonical Class Form

• **A default constructor**
  This is used internally to initialize objects and data members when no other value is available

• **A copy constructor**
  This is used, among other places, in the implementation of call-by-value parameters

• **An assignment operator**
  This is used to assign one value to another

• **A destructor**
  This is invoked when an object is deleted
Metaclasses

In Smalltalk (and Objective-C) classes are just objects, instances of class `Class`.

`new` is just a message given to a class object.

If we want to create constructors, where do we put them?

- They can't be part of the collection of messages of instances of the class, since we don't yet have an instance.
- They can't be part of the messages understood by class `Class`, since not all classes have the same constructor message.

Where do we put the behavior for individual class instances?
The solution is to create a new class, who's only instance is itself a class.

An elegant solution that maintains the simple instance/class relationship.
Chapter Summary

In this chapter we have examined the following topics:

- Message Passing Syntax
- Object Creation and Initialization (constructors)
- Accessing the Receiver from within a method
- Memory Management or garbage collection
- Metaclasses in Smalltalk