Fields accesses

- Field (instance variable)
- Let c be an instance of class $Circle$ and d be an instance of class $ColoredCircle$.

  $c.radius$ accesses the $radius$ field defined in $Circle$; $d.radius$ also accesses the $radius$ field defined in $Circle$ (by inheritance); and $d.color$ accesses the $color$ field defined in Circle.
- **How do we know which field is accessed?**

  Say we want to access the field f of an object of class X.

  We first check if f is defined in X itself; if not, we check if f is defined in the superclass of X (say, X'); if not, we check if f is defined in the superclass of X'; and so on, until we reach the root of the class hierarchy. If f is defined nowhere in this path, we know there is an error. In effect, we find the field f that is defined closest to the class X on the path from X to the root of the class hierarchy.
- This process of resolving which field is accessed can be done at compile time, or at run time.

JAVA

- **Field access:**

  In Java, field accesses (instance variable accesses) are resolved at compile time.
- **Method access:**

  In contrast to fields (instance variables), Java resolves method invocations at run time.

Example

- ```
  Circle c1;
  ColoredCircle d1;

  c1 = new Circle();
  d1 = new ColoredCircle();

  c1.draw(); -- What draw is it?

  c1 = d1;

  d1.draw(); -- What draw is it?
  c1.draw(); -- What draw is it?
  ```
- When we try to find out which method we mean by the last $c1.draw()$, we realize that although the variable $c1$ may be of declared to be of type $Circle$, it actually refers (at run time) to an object in class $ColoredCircle$. Hence, we search for an appropriate method starting from $ColoredCircle$.

Fields versus methods

- **Why are fields and methods treated differently in Java?**
- **Advantage of checking the resolving method invocations at run time.**

  Consider an array a that can hold $Circle$ objects. Since each $ColoredCircle$ is also a $Circle$, some elements of a may actually be $ColoredCircles$. Now if we want to draw all the objects in a, we can simply do $a[i].draw()$ for each element i in the array.

  If we resolve method invocations dynamically, we will invoke the method most appropriate for the objects that are actually stored in the array. In contrast, if we resolve method invocations at compile time, we will always invoke the $draw()$ method of $Circle$: rendering all circles without color!
- **Why are field accesses resolved at compile time?**

  – Field accesses are more common, and it becomes expensive to search for the appropriate field every time we want to access some data stored in an object.
  ```

– We can always write accessor methods for each field (a get- and a put- method for each field: to get its value and to store a new value respectively) and use these methods whenever we want to simulate run-time field resolution.

Information hiding

- From an engineering point of view separation of specification from implementation is essential.

  An important part of this separation is implemented by hiding some of the implementation details from the outside world. This way, external code cannot misuse the internal details.

- The most important visibility levels are

  – **private:** meaning that the field/method is completely hidden in the class, and not visible to anyone (even its subclass!) in the outside world.

  – **public:** meaning that the field/method is visible to the entire outside world.

  Most languages define other visibility levels that fall between these two extremes. For instance, Java's default visibility, as well as a level called **protected** fall between private and public in terms of visibility.

Object allocation in Java

- `Circle c1;`

  In Java, we have declared a variable $c1$ that is capable of referring to an object of class $Circle$.

  The declaration does not by itself make $c1$ refer to any object; it simply means that whatever $c1$ holds eventually will be a reference to an object in class $Circle$. The default value for variables such as $c1$ is the $null$ reference.

- Objects themselves are created using Constructor methods and the new key word. For instance to create a new object of class $Circle$, we say "$new\ Circle()$". This returns a reference to a freshly created $Circle$ object, which can then be assigned to $c1$.

Constructors in Java

- Each class is associated with a default constructor method: one that creates an instance of that class and initializes its fields to their respective default values. Users may define other constructor methods or override the default.

- In Java, programmers do not explicitly free memory at all. The system figures out which objects are accessible to the program, and releases the memory associated with all objects that are no longer accessible to the program (garbage collection).

Example

- ```
  Circle c1;
  ColoredCircle d1;
  c1 = new Circle();
  d1 = new ColoredCircle();

  c1 = d1;
  ```

- The object that was created using "$new\ Circle()$" is no longer accessible: its only reference $c1$ has been overwritten with the reference to the object created using "$new\ ColoredCircle()$".

  The Java run time system (JVM or any compiled version) detects such inaccessible objects and automatically releases the resources associated with them, using a technique called "garbage collection".

Parameter Passing

- Consider the following code fragment in a hypothetical language that allows global variables:

  ```
  int i;

  int f(int j)
  {
     i = 5;
     return j;
  }

  int g()
  {
    i = 3;
    return f(i);
  }
  ```

  Given the above definition, what will be the effect of calling $g()$?

- $g()$ calls $f(i)$ with i set to 3. If we follow the "call-by-value" parameter passing convention, the call to f will look like $f(3)$; this means that at the entry to f, the parameter j will be set to 3. Thus f returns 3.

- Of we follow the "call-by-reference" parameter passing convention, the reference to i and not the value of i will be passed to f. Thus, the parameter j in f will be set to the same entity referred by i. Now setting i to 5 silently changes the value of j too! Thus f returns 5.

Parameter passing

- C uses call-by-value on all parameters. If we want to pass any parameter by reference, we have to explicitly create references and do dereferencing at appropriate places in the code using the address-of ("&") and dereference operators ("*") respectively. By the way, C functions do not tolerate structure or array values as parameters or return values: one has to explicitly convert them into references and simulate call-by-reference for such values.

- Pascal supports both call-by-reference and call-by-value conventions: the programmer chooses which parameters should be passed by value and which by reference.

- In Java, all base objects (integers, floats, etc) are passed by value, while all other objects (including all user-defined objects) are passed by reference. Unlike C however, the programmer does not have to determine which ones should be passed by reference.
  ```