Lecture 6

We discuss, in detail, two of Java's four kinds of variable: local variables and formal parameters. We discuss the notions of variable declaration, initialisation, lifetime and identifier scope.

We see examples in our gibbering wreck class definition.

Java variables

- **Variables** are named storage locations. In Java, each variable has a **type** as well as a name (**identifier**).
- There are four kinds of variable in Java:
  - local variables; formal parameters; instance variables; and class variables.
- They differ in:
  - the ways they are declared, initialised and used;
  - their lifetimes;
  - the scope of their identifiers; and
  - in the case of instance variables & class variables, their visibility.

Local variables: declaration

- Local variables are used when you need to store something *temporarily* during the body of a method.
- *Declaring* a variable is asking for the variable & giving info about it (e.g. identifier & type)
- Declaration of a local variable is done within a *block* of statements within a method (or constructor) body, e.g.

```
public void someMethod()
{  int n;
   ...
   Color c;
   ...
}
```


Local variables: initialisation

- *Initialisation* of a variable is giving it an initial value.
- You must explicitly initialise local variables. The *compiler* will check up on you!
- Either declare and initialise in a single statement (generally better)

  ```
  int n = 3;
  ```

  or declare first and initialise later:

  ```
  int n;
  n = 3;
  ...
  Color c;
  ...
  c = new Color(10, 10, 10);
  ...
  ```


Local variables: scope of identifier

- An *identifier* is the name of something (variable, method, class,. . .)
- Identifiers have a certain scope: the portion of the program over which the identifier successfully names the variable
- Java is **statically-scoped** (also referred to as lexically-scoped): the scope of each identifier can be determined from the text of the program (we don't have to think about program execution), and so checks can be made by the *compiler*.
- The scope of a local variable identifier extends from the declaration of the variable to the end of the block in which it is declared.

Where are the errors? Compile-time or run-time?

```
public class SomeClass
{
   public void method1()
   {  int v1;
      int v2;
      v2 = v1;
   }

   public int method2()
   {  int v1 = 0;
      if (...)
      {  int v2 = v1;
      }
      else
      {  v2 = 0;
      }
      return v2;
   }

   public int method3()
   {  int v1 = 0;
      if (...)
      {  int v1 = 4;
         v2 = v1;
      }
      else
      {  v1 = 2;
      }
      return v1;
   }
}
```

Local variables: lifetimes

- A variable's *lifetime* is the period of the execution of the program during which the variable exists in memory
  - this is a run-time issue
  - it concerns when the variable is created, and when it is destroyed
  - i.e. when the memory is allocated/set aside, and when it is released
- A block's *local variables*: created, during program execution, each time the declaration is encountered; destroyed at end of execution of the block

Formal parameters

- Suppose a message is sent. It contains some data: the **actual parameters** or **actual arguments**, e.g.:

```
w1.getMimicry("nahnah");
```

- The object that receives the message needs variables where it can store this data
  - the **formal parameters** or **formal arguments**
  - for temporary storage of actual parameters during the method body
  - very much like local variables, but declared & initialised differently

Formal parameters: declaration, initialisation, scope of identifier & lifetime

- Declaration: within a method (or constructor) header (and one or two other places):

```
public void someMethod(int n)
{  ...
}
```

- Initialisation: when the method is invoked, using the actual parameters
- Scope: the body of the method
- Lifetime: created, during execution of the program, whenever the method is invoked; destroyed at the end of executing the method.

Gibbering Wrecks again: the `getMimicry` instance method

```
/**
 * Returns a copy of the parameter
 * concatenated to itself.
 *
 * @param aMessage a string that will be
 * mimicked.
 * @return the parameter concatenated to
 * an uppercase version of itself.
 */
public String getMimicry(String aMessage)
{


}
```