COMP 110/L Lecture 8

Mahdi Ebrahimi

Slides adapted from Dr. Kyle Dewey

Outline

- `public` / `private`
- "Getters" and "Setters"
- `toString()` method
- Memory representation

Review

Coding a basic calculator program in 3 approaches

`BasicCalculator.java`

Procedural Programming (PP) (using methods)

`BasicCalculatorPP.java`

Object Oriented Programming (OOP) (using class and object)

`BasicCalculatorOO.java`

Procedural Programming Language	Object Oriented Programming Language
1. Program is divided into functions.	1. Program is divide into classes and objects.
2. The emphasis is on doing things.	2. The emphasis on data.
3. Poor modeling to real world problems.	3. Strong modeling to real world problems.
4. It is not easy to maintain project if it is too complex.	4. It is easy to maintain project even if it is too complex.
5. Provides poor data security.	5. Provides strong data Security.
6. It is not extensible programming language.	6. It is highly extensible programming language.
7. Productivity is low.	7. Productivity is high.
8. Do not provide any support for new data types.	8. Provide support to new Data types.
9. Unit of programming is function.	9. Unit of programming is class.
10. Ex. Pascal , C , Basic , Fortran.	10. Ex. C++ , Java , Python.

public/private

`public`

Means it can be accessed from anywhere

public

Means it can be accessed from anywhere

```
public class PublicClass {
   public int i;
   public PublicClass(int x) {
      i = x;
   }
   public void printI() {
      System.out.println(i);
   }
}
```

Example

- `PublicClass.java`
- `PublicClassMain.java`

`private`

Means it can be accessed from **only** within the class

private

Means it can be accessed from **only** within the class

```
public class PrivateClass {
   private int i;
   private PrivateClass(int x) {
      i = x;
   }
   private void printI() {
      System.out.println(i);
   }
}
```

Example

- `PrivateClass.java`
- `PrivateClassMain.java`

Why `public` / `private`?

- Intentionally allows / disallows certain interactions between objects
- Stove example: perhaps only the stove can turn its burner on - make it `private`
- Commonly used to force changes to instance variables to go through methods (much more predictable)

“Getters” and “Setters”

Getters

Methods that return the value of an instance variable.
Generally, the instance variable is `private`.

Getters

Methods that return the value of an instance variable.
Generally, the instance variable is `private`.

```
public class HasGetter {
   private int saved;
   public HasGetter(int x) {
      saved = x;
   }
   public int getSaved() {
      return saved;
   }
}
```

Example:

`HasGetter.java`

Setters

Methods that change the value of an instance variable.
The instance variable is generally `private`.

Setters

Methods that change the value of an instance variable.
The instance variable is generally `private`.

```
public class HasSetter {
   private int saved;
   public HasSetter(int x) {
      saved = x;
   }
   public void setSaved(int to) {
      saved = to;
   }
}
```

Example:
`HasSetter.java`

Getter / Setter Purpose

- Access to instance variables forced to occur only via `get*` and `set*` methods
- These are the **only** points where change can occur
  - Much easier to predict and debug

`toString()` **Method**

`toString()`

Method used to convert an object to a `String`.

Called automatically in `String` contexts.

`toString()`

Method used to convert an object to a `String`.
Called automatically in `String` contexts.

```
public class HasToString {
   private String held;
   public HasToString(String s) {
      held = s;
   }
   public String toString() {
      return held;
   }
}
```

Example:
`HasToString.java`

Memory Representation

On `new`

Each use of `new` creates a new object in memory.

On `new`

Each use of `new` creates a new object in memory.

```
new Foo();
new Bar();
```

On `new`

Each use of `new` creates a new object in memory.

```
new Foo();
new Bar();
```

In Memory

What `new` Returns

- `new` returns a *reference* to the created object
- References can be copied just like `int`, `double`, etc.
- Copying a reference does **not** copy the underlying object

-This is the difference between copying a house and copying an address.
- References act like addresses (and some languages even call them addresses!)

What `new` Returns

- `new` returns a *reference* to the created object
- References can be copied just like `int`, `double`, etc.
- Copying a reference does **not** copy the underlying object

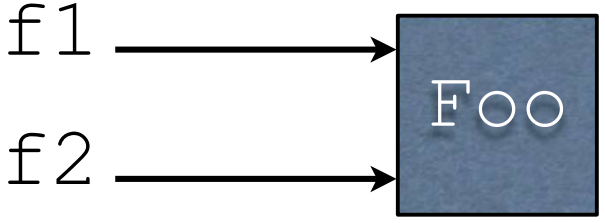
```
Foo f1 = new Foo();
Foo f2 = f1;
```

-This is the difference between copying a house and copying an address. References act like addresses (and some languages even call them addresses!)

What `new` Returns

- `new` returns a *reference* to the created object
- References can be copied just like `int`, `double`, etc.
- Copying a reference does **not** copy the underlying object

```
Foo f1 = new Foo();
Foo f2 = f1;
```



-This is the difference between copying a house and copying an address. References act like addresses (and some languages even call them addresses!)