# Primitive Type Variables
# Object Reference Variables
# Local Variables

CSCU9A3
Data Structures, Objects and Algorithms

David Cairns

Derived from *Big Java* by Cay Horstmann
John Wiley & Sons

## Object References

- Object Reference: Record the location of an object
  - The `new` operator returns a reference to a new object:

```
Rectangle box = new Rectangle();
```

- Multiple object variables can refer to the same object:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
Rectangle box3 = box2;
box2.translate(15, 25);
```

# Object References

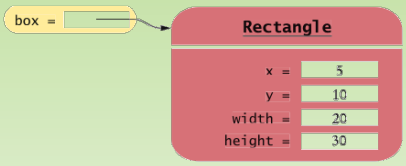



**Figure 17** An Object Variable Containing an Object Reference

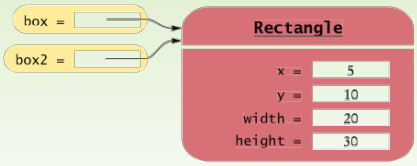



**Figure 18** Two Object Variables Referring to the Same Object



# Objects

- We can have two different objects of the same class.
  - We use variables to store references to each of the objects.
  - If we lose track of a reference, we lose the object (it becomes garbage, in C++ this would be a memory leak)

```
String s1 = new String("Hello World");
String s2 = new String("Missing car");
s2 = s1;
// At this point we have lost track of the object containing "Missing car"
```

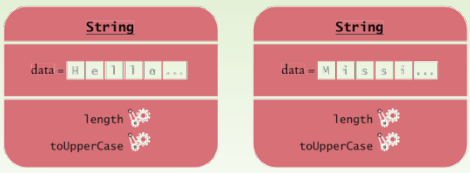



**Figure 5** A Representation of Two String Objects

# Primitive Type Variables vs Object Variables

- Primitive Type Variables ≠ Object Variables
  - This can cause a lot of confusion…
  - Primitive type – int, float, double, boolean, char
    - Stores the raw value, not a reference to where the value is stored

```
int x = 3;
float pi = 3.1459;
double bigPi = 3.141592653589793;
char lastCharacter = 'z';
boolean open = true;
```

    - Note
      - No mention of 'new'
      - The statement:

```
int y = x;
```

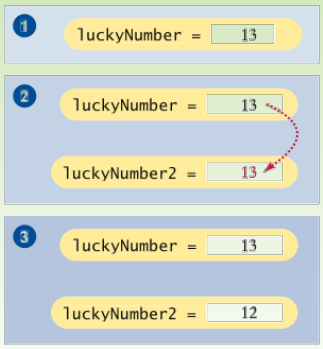
        would cause the value of x to copied into y. If you then changed x to 7, it would not change the contents of y.



# Primitive Type Variables: Example

```
int luckyNumber = 13;
int luckyNumber2 = luckyNumber;
int luckyNumber2 = 12;
```

**Figure 20**
Copying Numbers



# Object Reference Variable: Example

```
Rectangle box = new Rectangle(5,10,20,30);
Rectangle box2 = box;
box2.translate(15,25);
```

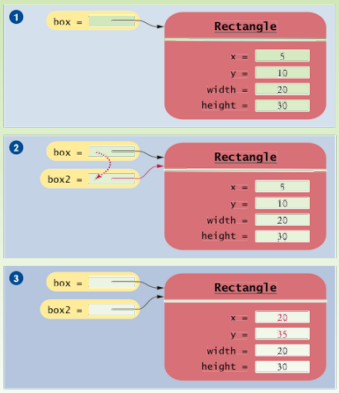



**Figure 21** Copying Object References



# Instance Variables

- **Instance variables** are used to store the attributes of an object
  - **Instance** of a class: an object of the class
    - e.g. for the class Car, one instance could be a red Ferrari, another instance could be a black Porsche
- The class declaration specifies the instance variables:
  - Previously we used make, model, colour etc for Car
  - What about a Counter class, used to count things…

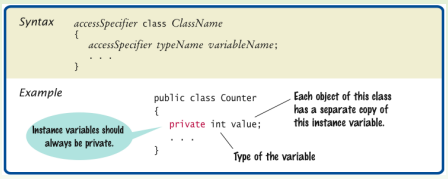
```
public class Counter
{
  private int value;
  …
}
```

# Instance Variables

- An instance variable declaration consists of the following parts:

| | |
|---|---|
| *access specifier* | `private, public` |
| *type of variable* | `int, float, String` |
| *name of variable* | `value, colour` |

  - Each object of a class has its own set of instance variables
  - You should declare all instance variables as private






# Instance Variables: Example

```
concertCounter = new Counter();
boardingCounter = new Counter();
```

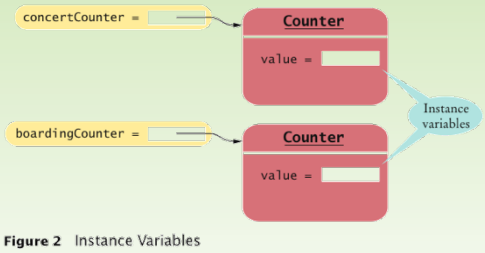



**Figure 2** Instance Variables

## Accessing Instance Variables

- The `count` method advances the counter value by 1:

```
public void count()
{
    value = value + 1;
}
```

- The `getValue` method returns the current value:

```
public int getValue()
{
    return value;
}
```

- Private instance variables can only be directly accessed by methods of the same class



## Encapsulation

- Encapsulation is the process of hiding object data and then providing methods for controlling access to the data
  - To encapsulate data, declare instance variables as private and declare public methods that access them.
- Encapsulation allows a programmer to use a class without having to know its implementation. The internal structure is hidden.
  - Information hiding makes it simpler for the implementor of a class to locate errors and change implementations
  - Protects your data (get/set methods accessor/mutator methods) from others and from yourself…
  - What other approaches could be used to implement the class Counter?
    - As a user of this class, would you care?
  - Encapsulation allows you to implement a class and then improve the efficiency of the implementation without affecting the user's code.

# Local Variables

- Local and parameter variables belong to a method
  - When a method or constructor runs, its local and parameter variables come to life
  - When the method or constructor exits, the local and parameter variables are lost
- Instance variables belong to an object, not methods
  - When an object is constructed, its instance variables are created
  - The instance variables stay alive until the object they belong to is no longer referenced
  - In Java, the garbage collector periodically reclaims objects when they are no longer referenced
    - In C++, you have to work out yourself what is not needed and 'free' it.
      - Memory leaks occur if you lose track and don't free it.
      - Even worse, you can free up memory referred to by something you haven't actually finished using. You often don't realise until a lot later that your memory is corrupted and you try to use the object associated with it.