COMP 202 - Week 7

- Now, we will revisit objects in more detail, as well as introduce the concept of polymorphism (covered thoroughly in week 11) through interfaces.

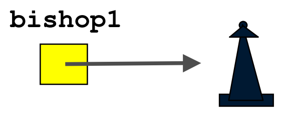
- This week we focus on:
  - object references and aliases
  - passing objects as parameters
  - Interfaces

References

- Recall that an object reference holds the memory address of an object

- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object

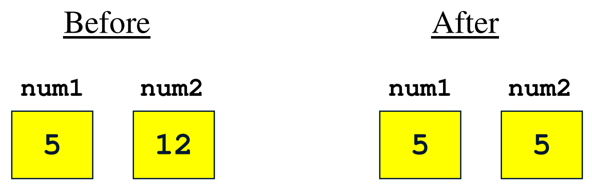
```
ChessPiece bishop1 = new ChessPiece();
```



Assignment Revisited

- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

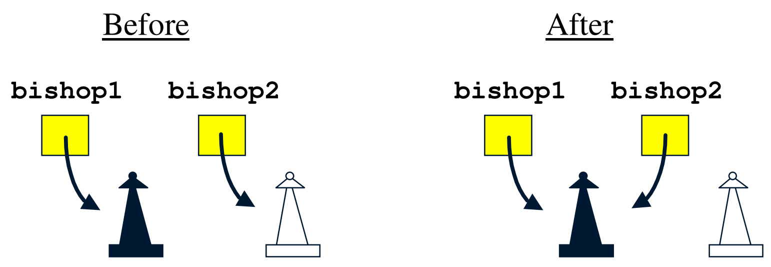
```
num2 = num1;
```



Reference Assignment

- For object references, assignment copies the memory location:

```
bishop2 = bishop1;
```



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- One object (and its data) can be accessed using different variables
- Aliases can be useful, but should be managed carefully
- Changing the object’s state (its variables) through one reference changes it for all of its aliases

Garbage Collection

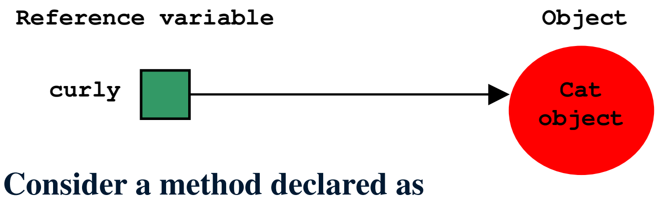
- When an object no longer has any valid references to it, it can no longer be accessed by the program
- It is useless, and therefore called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In some other languages, the programmer has the responsibility for performing garbage collection

Passing Objects to Methods

- Parameters in a Java method are *passed by value*
- This means that ***a copy*** of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- See ParamPassTest.java
- Passing parameters is essentially an assignment
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Parameter Passing

- In a Java statement such as `Cat curly = new Cat();` the variable `curly` is not an object, it is simply a reference to an object (hence the term reference variable).



- Consider a method declared as

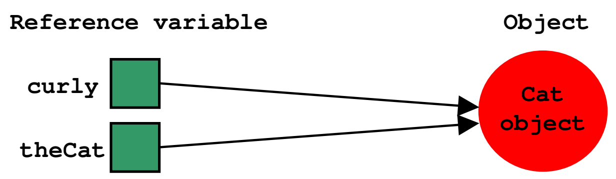
```
public void veterinarian(Cat theCat) {...}
```

- If we call this method passing in a reference to a Cat object, what happens exactly?

```
Cat curly = new Cat();
veterinarian(curly);
```

Parameter Passing

- The value of the variable `curly` is passed *by value*, and the variable `theCat` within `veterinarian()` receives a copy of this value.
- Variables `curly` and `theCat` now have the same value.
- However, what does it mean to say that two reference variables have the same value ?
- It means that both variables refer to the same object:



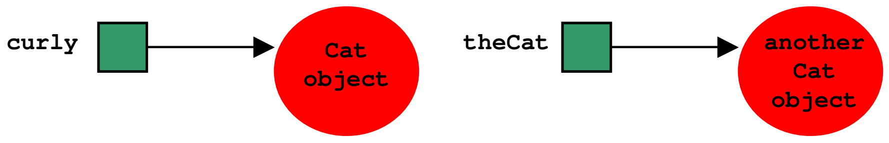
- Within `veterinarian()` you can now update the `Cat` object via variable `theCat`.

Parameter Passing

- An object can have multiple references to it.
- In this example, we still have just the one object, but it is being referenced by two different variables.
- But if you change the value of the variable `theCat` within `veterinarian()` so that it refers to a different object:

```
theCat = new Cat(4,3.8f,false,aHouse);
```

- Then the value of variable `curly` within the calling method remains unchanged, and variable `curly` will still refer to the same `Cat` object that it always did:



- See PassByValue.java , Cat.java

Passing Objects to Methods

- What you do to a parameter inside a method may or may not have a permanent effect (outside the method)

- See ParameterPassing.java
- See ParameterTester.java
- See Num.java

- Note the difference between changing the reference and changing the object that the reference points to

Interfaces

- A Java *interface* is a collection of abstract methods and constants

- An *abstract method* is a method header without a method body

- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, it is usually left off

- An interface is used to formally define a set of methods that a class will implement

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it must define all methods in the interface or the compiler will produce errors.

Interfaces

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interfaces

- A class that implements an interface can implement other methods as well

- See Predator.java
- See Lion.java
- See Hyena.java
- See Leopard.java

- A class can implement multiple interfaces
- The interfaces are listed in the implements clause, separated by commas
- The class must implement all methods in all interfaces listed in the header

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Doable obj;
```

- The `obj` reference can be used to point to any object of any class that implements the `Doable` interface
- The version of `doThis` that the following line invokes depends on the type of object that `obj` is referring to:

```
obj.doThis();
```

Polymorphism via Interfaces

- That reference is *polymorphic*, which can be defined as "having many forms"
- That line of code might execute different methods at different times if the object that `obj` points to changes
- See Hunting.java
- Note that polymorphic references must be resolved at run time; this is called *dynamic binding*
- Careful use of polymorphic references can lead to elegant, robust software designs

Interfaces

- The Java standard class library contains many interfaces that are helpful in certain situations
- The `Comparable` interface contains an abstract method called `compareTo`, which is used to compare two objects
- The `String` class implements `Comparable` which gives us the ability to put strings in alphabetical order
- The `Iterator` interface contains methods that allow the user to move through a collection of objects easily