Computer Notebook Example

Has-a Relationship

```
public class Computer {
  private Memory mem;
  ...
}

public class Memory {
  private int size;
  private int speed;
  private String kind;
  ...
}
```

A Computer has only one Memory

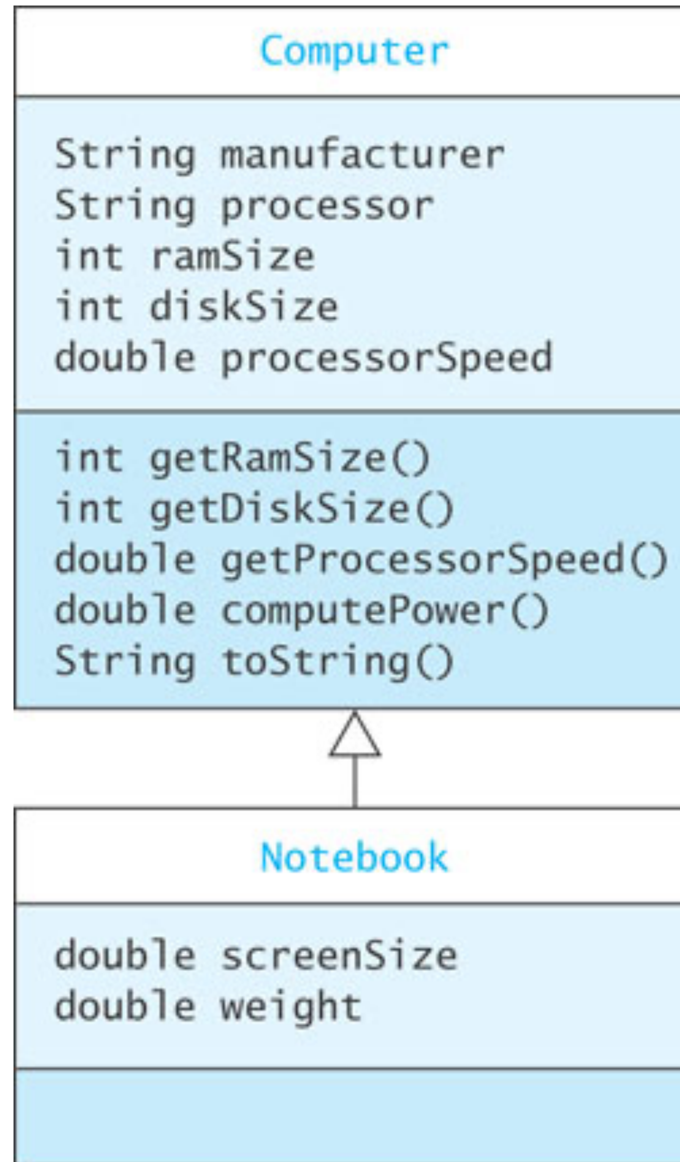
But a Computer is not a Memory (i.e. not an is-a relationship)

If a Notebook extends Computer,then the Notebook is-a Computer

Has-a can contain more than one. A computer could have two types of memory, RAM and Video memory.

You can also need to consider, would the Computer be a valid entity without Memory? That is, is the part optional?

Is-a Relationship



Method Overriding

- if we declare and then run:

```
Computer myComputer = new Computer(
   "Acme", "Intel", 2, 160, 2.4);

Notebook yourComputer = new Notebook(
   "DellGate", "AMD", 4, 240, 1.8, 15.0, 7.5);

System.out.println("My computer is:\n" +
  myComputer.toString());

System.out.println("Your computer is:\n" +
   yourComputer.toString());
```

Method Overriding

- the output would be:

```
My Computer is:
Manufacturer: Acme
CPU: Intel
RAM: 2.0 gigabytes
Disk: 160 gigabytes
Speed: 2.4 gigahertz

Your Computer is:
Manufacturer: DellGate
CPU: AMD
RAM: 4.0 gigabytes
Disk: 240 gigabytes
Speed: 1.8 gigahertz
```

- The screensize and weight variables are not printed because Notebook has not defined a toString() method, so it uses the one it **inherited** from Computer

Method Overriding

- To define a toString() for Notebook:

```
public String toString() {
  String result = super.toString() +
           "\nScreen size: " +
           screenSize + " inches" +
           "\nWeight: " + weight +
           " pounds";
  return result;

}
```

super.methodName()

Using the prefix super in a call to a method methodName calls the method with that name in the superclass of the current class

- Now Notebook's toString() method will override Computer's toString() and will be called for all Notebook objects.

Method Overloading vs Overriding

- A method which has the same **name, return type, and parameters** as a method in a superclass will override the superclass' version of the method
- Methods with the **same name but different parameters** are *overloaded*

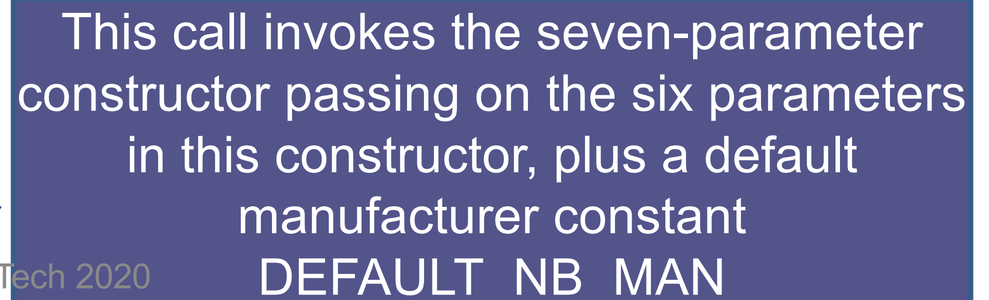
Method Overloading

- Notebook constructor:

```
public Notebook(String man, String processor,
               double ram, int disk, double procSpeed,
               double screen, double wei)
{
    . . .
}
```

- To add a default manufacturer for a Notebook, create a constructor with six parameters instead of seven:

```
public Notebook(String processor, double ram, int disk,
               double procSpeed, double screen, double wei)
{
      this(DEFAULT_NB_MAN, processor, ram, disk,
procSpeed,
           screen, wei);
}
```



Method Overriding: Pitfall

- When ***overriding*** a method, the method must have the same name and the same number and types of parameters in the same order
- If not, the method will be ***overloaded***
- The annotation @Override preceding an overridden method will signal the complier to issue an error if it does not find a corresponding method to override

```
@Override
public String toString() {
  . . .
}
```

Revisiting Inheritance and Interfaces

- Inheritance and Interfaces are two approaches for allowing an instance of a class to have multiple types
- **<u>Polymorphism</u> is when the type of the instance is determined at runtime**
  - **At compile time the variable type is checked to make sure that any methods invoked actually exist.**
  - **At runtime the JVM looks at the actual object's type to determine which method implementation to call.**

CS2-ExPolymorphismComputer

- For example, if you write a program to reference computers, you may want a variable to reference a Computer or a Notebook
- If you declare the reference variable as

  ```
  Computer theComputer;
  ```

  it can reference either a Computer or a Notebook,

  because a Notebook is-a Computer

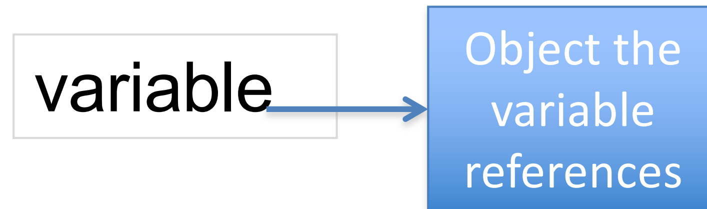
  i.e. either is valid

  theComputer = new Computer(…);

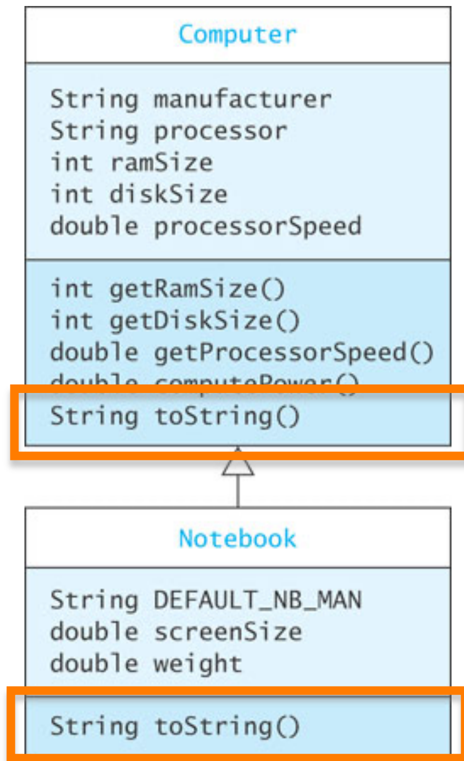
  theComputer = new Notebook(…);

Polymorphism Example -Visual

- Remember:



- Because Notebook is inherited from Computer, these are both valid:

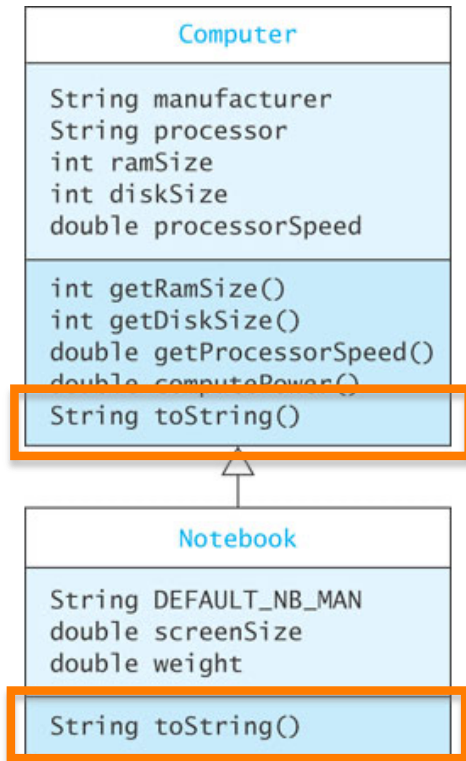


Object Types Are Determined at Runtime

- Given:

```
Computer theComputer;
theComputer = new Notebook("Bravo","Intel", 4,
                                    240, 2/4, 15.07.5);
System.out.println(theComputer.toString());
```

- Which toString() method will be called, Computer's or Notebook's?



Object Types Are Determined at Runtime

- The JVM correctly identifies the type of theComputer as Notebook and calls the toString() method associated with Notebook
- This is an example of polymorphism
- The type cannot be determined at compile time, but it can be determined at run time

Benefit of OO

- Declare variable and parameters as the more general type(interface or parent class) and store specialized versions (subclasses) but you don’t need to program checks on the type to call different methods accordingly, JVM determines this.
- This improves **extensibility.** New types — subclasses or alternate interface implementations — can be added later, without having to the change code that references only the interface or parent class types.