LECTURE 7

Object-Oriented Programming and Class Hierarchies

Method Overriding (cont.)

- To define a `toString()` for `Notebook`:

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

}
```

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

Method Overriding (cont.)

- 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

- No
  ov
  `toString()` method and will be called for all `Notebook` objects

Method Overloading (cont.)

- Methods in the class hierarchy which have the same name, return type, and parameters *override* corresponding inherited methods
- Methods with the same name but different parameters are *overloaded*

Method Overloading (cont.)

- Take, for example, our Notebook constructor:

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

- If we want to have a default manufacturer for a Notebook, we can 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, double ram, int disk, double procSpeed,
        double screen, double wei);
}
```

Method Overloading: 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 *overload*
- This error is common; 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()
{
  . . .
}
```

- It is good programming practice to use the `@Override` annotation in your code

Polymorphism

- Polymorphism means *having many shapes*
- Polymorphism is a central feature of OOP
- It enables the JVM to determine at run time which of the classes in a hierarchy is referenced by a superclass variable or parameter

Polymorphism (cont.)

- 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`

Polymorphism (cont.)

- Suppose the following statements are executed:

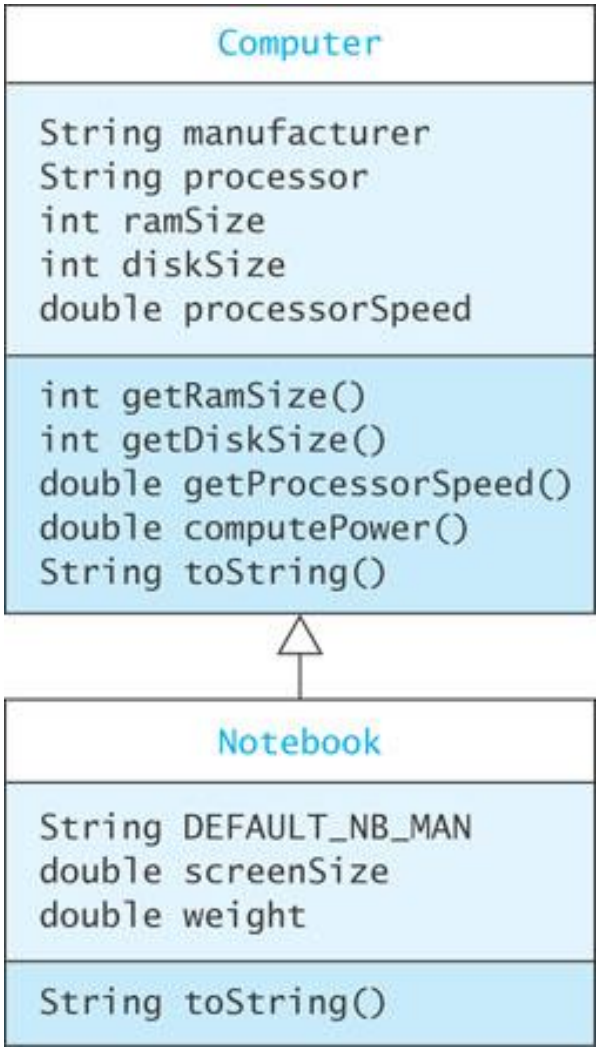
```
theComputer = new Notebook("Bravo", "Intel",
  4, 240, 2.4, 15, 7.5);
System.out.println(theComputer.toString());
```

- The variable `theComputer` is of type `Computer`

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

Polymorphism (cont.)

- 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* (why?)



Polymorphism (cont.)

- `Computer [] labComputers = new Computer[10];`
- `labComputers[i]` can reference either a `Computer` or a `Notebook` because `Notebook` is a subclass of `Computer`
- For `labComputers[i].toString()` polymorphism ensures that the correct `toString` method will be executed

Methods with Class Parameters

- Polymorphism simplifies programming when writing methods with class parameters
- If we want to compare the power of two computers (either `Computers` or `Notebooks`) we do not need to overload methods with parameters for two `Computers`, or two `Notebooks`, or a `Computer` and a `Notebook`
- We simply write one method with two parameters of type `Computer` and allow the JVM, using polymorphism, to call the correct method

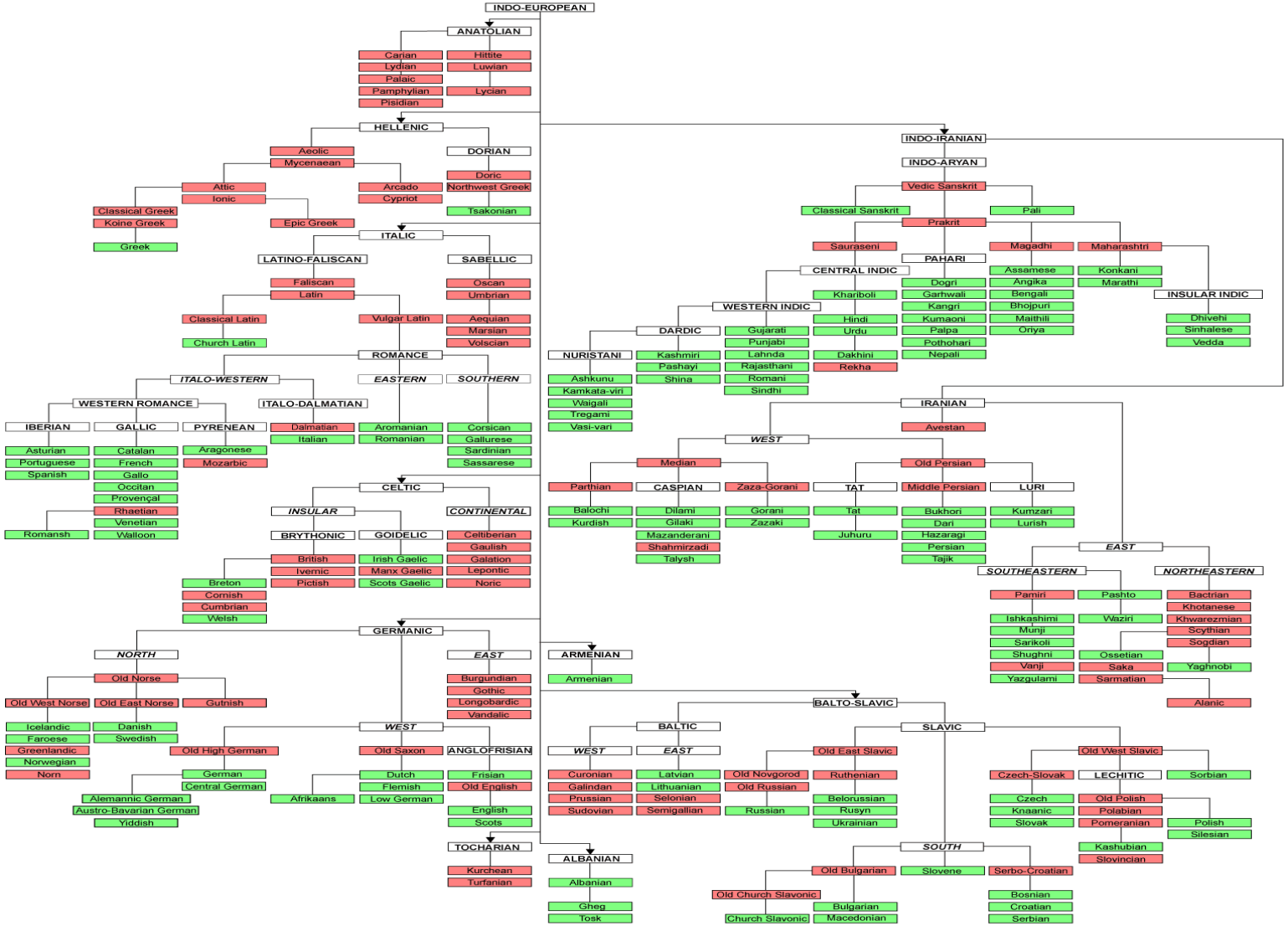
Methods with Class Parameters (cont.)


```
/** Compares power of this computer and its argument computer
    @param aComputer The computer being compared to this computer
    @return -1 if this computer has less power,
            0 if the same, and
            +1 if this computer has more power.
*/
public int comparePower(Computer aComputer)
{
    if (this.computePower() < aComputer.computePower())
            return -1;
    else if (this.computePower() == aComputer.computePower())
            return 0;
    else return 1;
}
```

Abstract Classes

Why are the abstract Classes needed?

An Abstract language: Indo-European



Abstract Classes

- An abstract class is denoted by using the word `abstract` in its heading:

```
visibility abstract class className
```

- An abstract class differs from an actual class (sometimes called a concrete class) in two respects:
  - An abstract class cannot be instantiated
  - An abstract class may declare abstract methods
- Just as in an interface, an abstract method is declared through a method heading:

```
visibility abstract resultType methodName (parameterList);
```

- A concrete class that is a subclass of an abstract class must provide an implementation for each abstract method

Needed to create ***actual*** subclasses!

Abstract Classes (cont.)

- Use an abstract class in a class hierarchy when you need a base class for two or more subclasses that share some attributes
- You can declare some or all of the attributes and define some or all of the methods that are common to these subclasses
- You can also require that the actual subclasses implement certain methods by declaring these methods abstract

Class `Object`

- `Object` is the root of the class hierarchy
- Every *class* has `Object` as a superclass
- All classes inherit the methods of `Object` but may override them

Method	Behavior
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.
`Class<?> getClass()`	Returns a unique object that identifies the class of this object.

Example of an Abstract Class


```
public abstract class Food {
  public final String name;
  private double calories;
  // Actual methods
  public double getCalories () {
    return calories;
  }
  protected Food (String name, double calories) {
    this.name     = name;
    this.calories = calories;
  }
  // Abstract methods
  public abstract double percentProtein();
  public abstract double percentFat();
  public abstract double percentCarbs();
}
```

Java Wrapper Classes

- A *wrapper class* is used to store a primitive-type value in an object type

Remember that an integer is not an object in Java!

Pre-defined in Java
Declares what the (concrete) subclasses have in common