Data Structures

OOP and Class Hierarchies

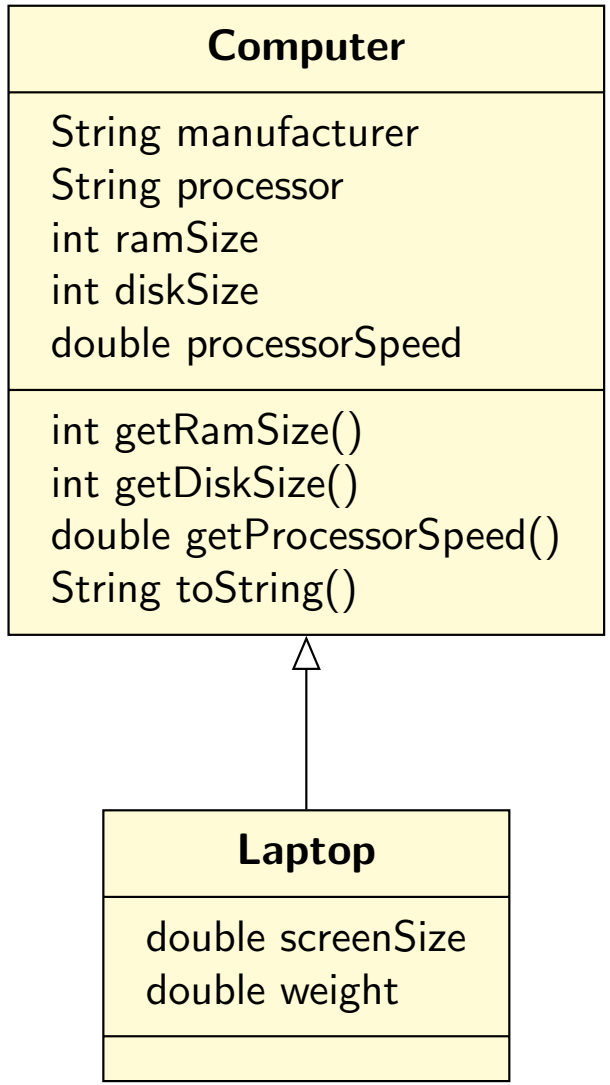
CS284

Objectives

- Method overriding
- Method overloading
- Polymorphism
- Abstract classes
- Casting in a class hierarchy

Method Overriding: Laptop vs. Computer

- Recall from last class
- The `toString` method for Computer does not print the values of screenSize and weight



Overriding `toString` method

Suppose we run:

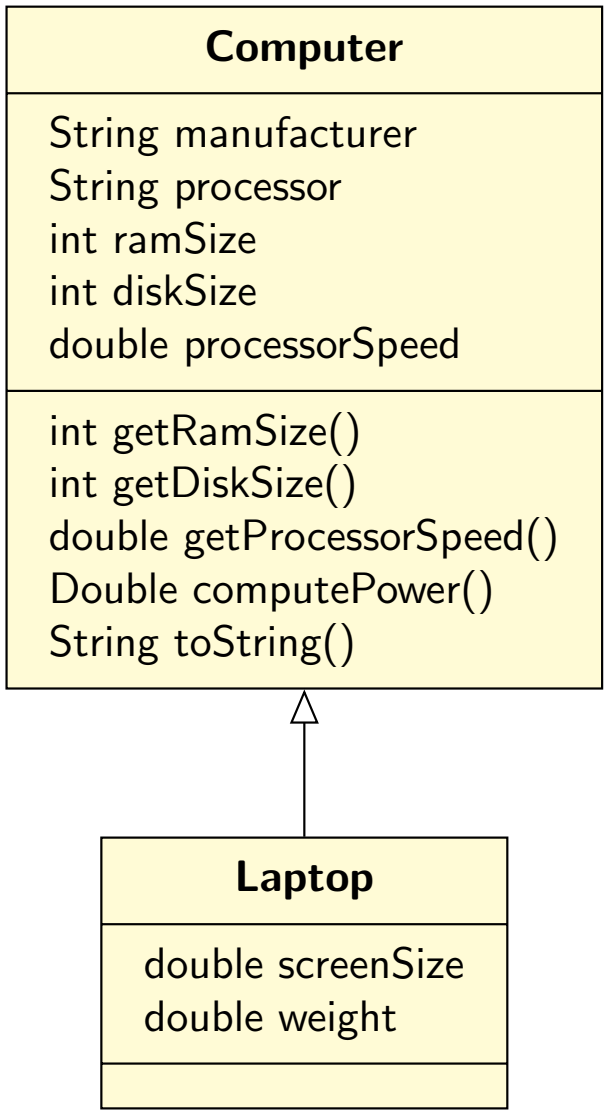
```
Computer computer = new Computer("Acme", "Intel", 2,
160, 2.4);

Laptop laptop = new Laptop("DellGate", "AMD", 4,
240, 1.8, 15.0, 7.5);

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

System.out.println("Laptop is:\n"
+ laptop.toString());
```

Method Overriding

The output is

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

Laptop 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 Laptop has not defined a toString() method



Method Overriding: Laptop vs. Computer

- Laptop can define its own `toString` method:

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

- Overrides Computer's inherited `toString()` method and will be called for all Laptop objects

Method Overriding

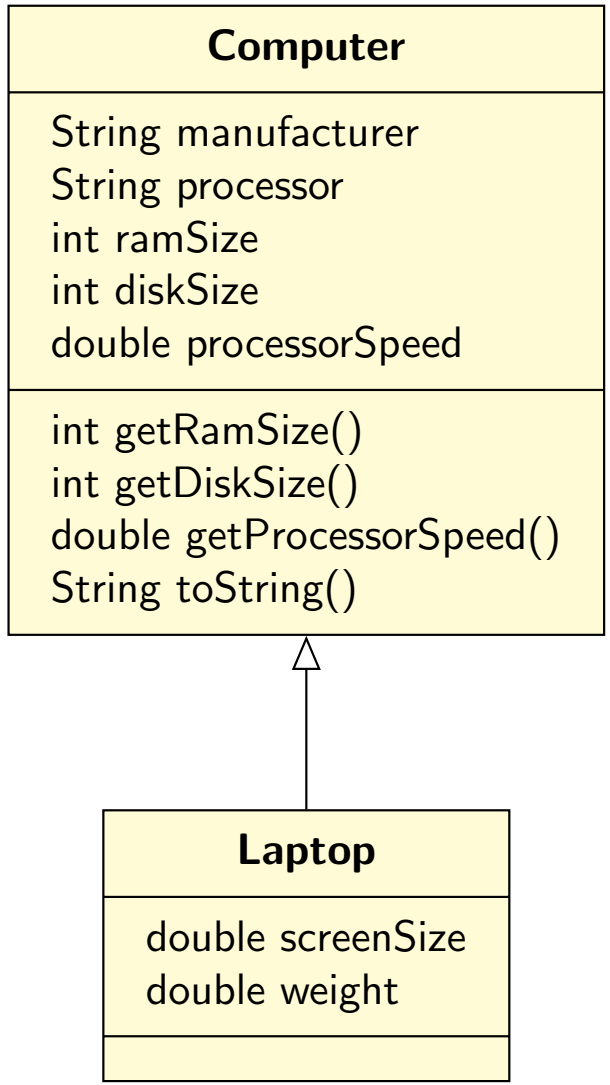
Run the following snippet of code again:

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

Laptop laptop = new Laptop("DellGate", "AMD", 4,
240, 1.8, 15.0, 7.5);

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

System.out.println("Laptop is:\n"
+ laptop.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
Screen size: 15.0
Weight: 7.5
```



Method Overriding: Summary

- A subclass define the same method differently from the parent class
- The method in a class which is *overriden* is no longer available
- Hence why we speak of "overriding"
- In order to override a method in a super class, a method in the subclass must share the same signature (i.e., name, return type, and parameters) as the super class

Method Overloading

- Having multiple methods with the same name but different signatures is called over*loading*
- Difference between overriding and overloading:
  - Overriding: methods with the same signature from two classes in a class hierarchy
  - Overloading: methods with the same name and *different* signatures *within* the same class

An Example: Overloading Constructors in Laptop

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

If we want to have a default manufacturer for a Laptop, we can create a constructor with six parameters instead of seven

```
public Laptop(String processor, double ram, int
disk, double procSpeed, double screen, double weight)
{
  this(DEFAULT_NB_MAN, double ram, int disk,
  double procSpeed, double screen, double weight)
}
```

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
- Failing to follow this rule will result in an accidental *overload*ing of the method (instead of the intended overriding)
- To avoid confusing overriding with overloading: the annotation @Override preceding an overridden method will signal the compiler 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 this annotation

Example: Overriding

```
public class A {

  public static void main(String[] args){
    A x;
    x=new B();
    System.out.print(x.m(5));
  }

  public int m(float x) {
     return 10; }
}
public class B extends A {
  public int m(float x) {
    return 20; }
}
```

Output: 20

Example: Overloading

```
public class A {
  public static void main(String[] args){
    A x;
    x=new B();
    System.out.print(x.m(5));
  }

  public int m(int x) {
     return 10; }
  }

  public class B extends A {
     public int m(float x) {
       return 20; }
}
```

Output: 10

Example #2: Overloading

```
public class A {
  public static void main(String[] args){
    A x;
    x=new B();
    System.out.print(x.m(5));
  }

  private int m(int x) {
     return 10; }
}
public class B extends A {
  public int m(int x) {
    return 20; }
}
```

Output: ?

Polymorphism

- Means having many shapes and is 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

Example

- If you write a program to reference computers, you may want a variable to reference a Computer or a Laptop
- If you declare the reference variable as

```
Computer theComputer;
```

  it can reference either a Computer or a Laptop—because a Laptop *is-a* Computer

Polymorphism

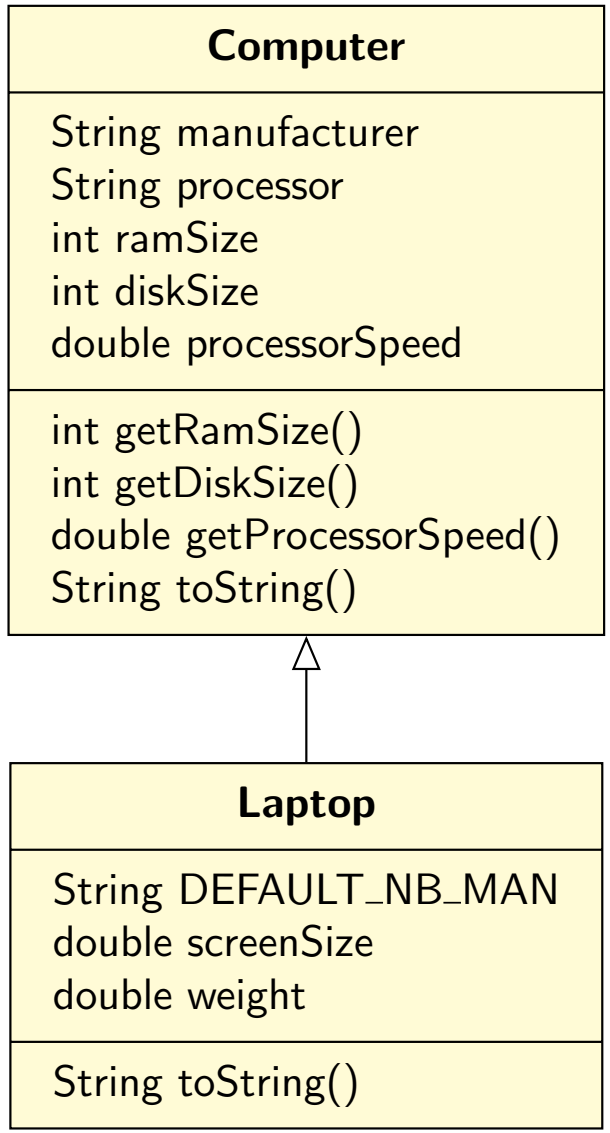
- Suppose the following statements are executed:

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

- The variable `theComputer` is of type `Computer`,
- Which `toString()` method will be called, Computer's or Laptop's?

Polymorphism

- The JVM correctly identifies the run time type of theComputer as Laptop and calls the toString() method associated with Laptop
- This is an example of polymorphism



Polymorphism

```
Computer[] labComputers = new Computer[10];
```

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

Example: Comparing Laptop with Computer

- If we want to compare the *computing power* of two computers (either Computers or Laptops) we do not need to overload methods with parameters for two Computers, or two Laptops, or a Computer and a Laptop
- We simply write one method with two parameters of type Computer and allow the JVM, using polymorphism, to call the correct method

```
/** Compute power of this computer
*/
public double computePower()
   { return ramSize * processorSpeed; }
```

Example: Comparing Laptop with Computer

- The following code is placed in the class `Computer`

```
/** Compares power of this comp. and its argument comp.
   @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;
}
```

Example: Comparing Laptop with Computer

- The following code is valid; note that the argument to `comparePower` is of type `Laptop`
- It prints 1

```
Computer c1 = new Computer("pc",7,8);
Laptop c2 = new Laptop("laptop",2,3);

System.out.println(c1.comparePower(c2));
```

- What happens without polymorphism?
  - We will have to double the LOC (lines of code)
  - Polymorphism improves the readability and code reuse

Abstract Classes

- Denoted by using the word **abstract** in its heading

```
public abstract class Food ...
```

- A concrete class *extends* an abstract class
- Differs from an actual class (sometimes called a concrete class) in two aspects:
  - An abstract class cannot be instantiated
  - An abstract class may declare *abstract methods*, i.e., not implemented
- Example of an abstract method:

```
public abstract double percentFat();
```

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

Abstract Classes

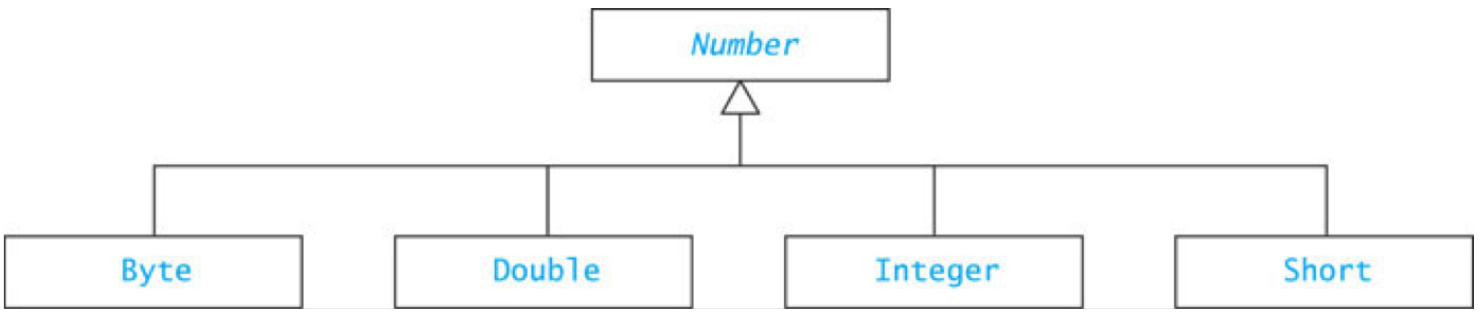
- Use an abstract class in a class hierarchy when you need a base class for two or more subclasses that share some attributes
- You can declare some or all of the attributes and define some or all of the methods that are common to these subclasses
- You can also require that the actual subclasses implement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

Example: Food

```
public abstract class Food {
  public final String name;
  public double calories;
  // Actual methods
  public double getCalories () {
    return calories;
  }
  public 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();
}
```

Example: Number

- A wrapper class is used to store a primitive-type value in an object type
- The `Number` class is an example of an abstract class too
- It relates the following wrapper classes



Abstract Classes vs. Interfaces

- A Java interface can
  - Declare methods, but cannot implement them
  - These methods are called abstract methods.
  - All fields are automatically public, static, and final
- An abstract class can have:
  - abstract methods
  - concrete methods
  - data fields
- Abstract classes and Interfaces cannot be instantiated
- Interfaces: allow multiple inheritance, (abstract) classes do not
- Abstract classes: allow code to be shared, interfaces do not

Abstract Classes vs. Interfaces

- An abstract class can have constructors!
  - Purpose: initialize data fields when a subclass object is created
  - The subclass uses `super(...)` to call the constructor
- An abstract class may implement an interface, but need not define all methods of the interface
  - Implementation is left to subclasses
  - Demo: Abstract_class.java

Inheriting from Interfaces vs. Classes

- A class can *extend* 0 or 1 superclass
- An interface cannot *extend* a class
- A class can *implement* 0 or more interfaces

How to fix the error in the following code?

```
 abstract class A {
private abstract String B();
}
```

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

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 the object

Method `toString`

- You should always override toString method if you want to print the object's state
- If you do not override it:
  - Object.toString will return a String
  - Just not the String you want!
- Example: ArrayBasedPD@ef08879
- The name of the class, @, instance's hash code

Type of Reference vs. Type of Object Referenced

- As shown previously with Computer and Laptop, a variable can refer to object whose type is a subclass of the variable's declared type

```
Object aThing = new Integer(25);
```

- The compiler always verifies that a variable's type includes the class of every expression assigned to the variable (e.g., class Object must include class Integer)
- In the above example:
  - Type of reference: Object;
  - Type of object referenced: Integer;

Operations Determined by Type of *Reference*

```
Object aThing = new Integer(25);
```

- Is `aThing.intValue()` valid?
- No, because `Object` does not have an `intValue()` method (even though `Integer` does, the reference is considered of type `Object`)
- That is, the validity of operations is determined by the type of *reference* instead of the type of *referenced object*