Introduction to Java Programming

Polymorphism

CS501

Objectives

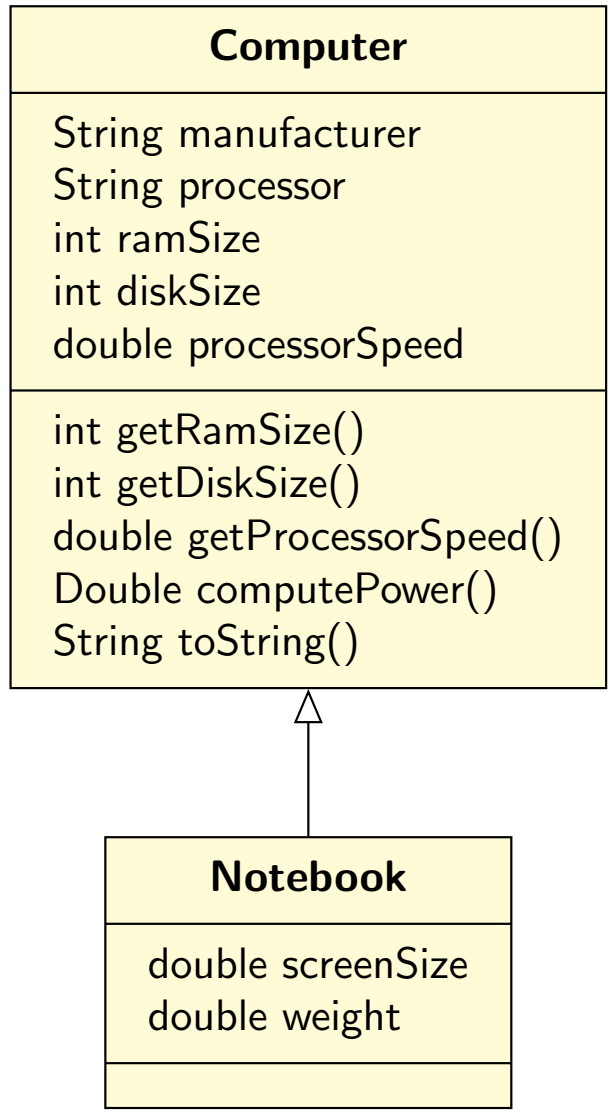
- How Java determines which method to execute when there are multiple methods
- Abstract classes
- Abstract data types and interfaces
- Object class and overriding Object class methods
- Exception hierarchy – out of scope
- Packages and visibility
- Class hierarchy for shapes

Method Overloading

- Methods in the class hierarchy which have the same name, return type, and parameters override corresponding inherited methods
- The method in a class which is overriden by one in the subclass is no longer available
- Hence why we speak of "overriding"

Method Overriding

- Recall from last class



Method Overriding

Suppose we 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());
```

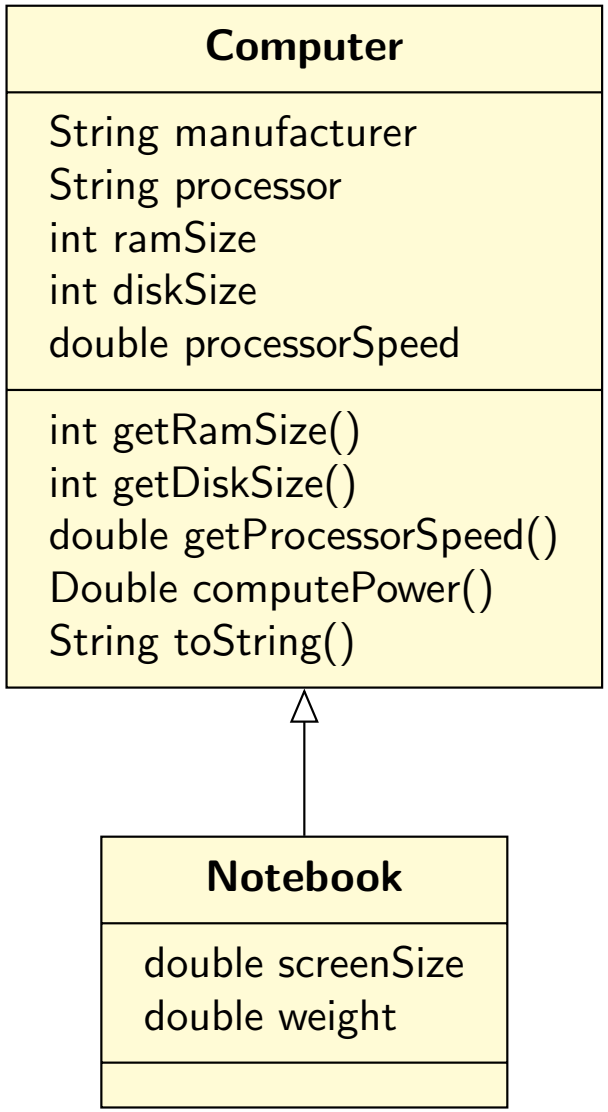
Method Overriding

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



Method Overriding

- In Notebook:

```
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 Notebook objects
  - **`super`**`.methodName()` calls the method with that name in the superclass of the current class

Method Overriding

Suppose we now run again the snippet of code:

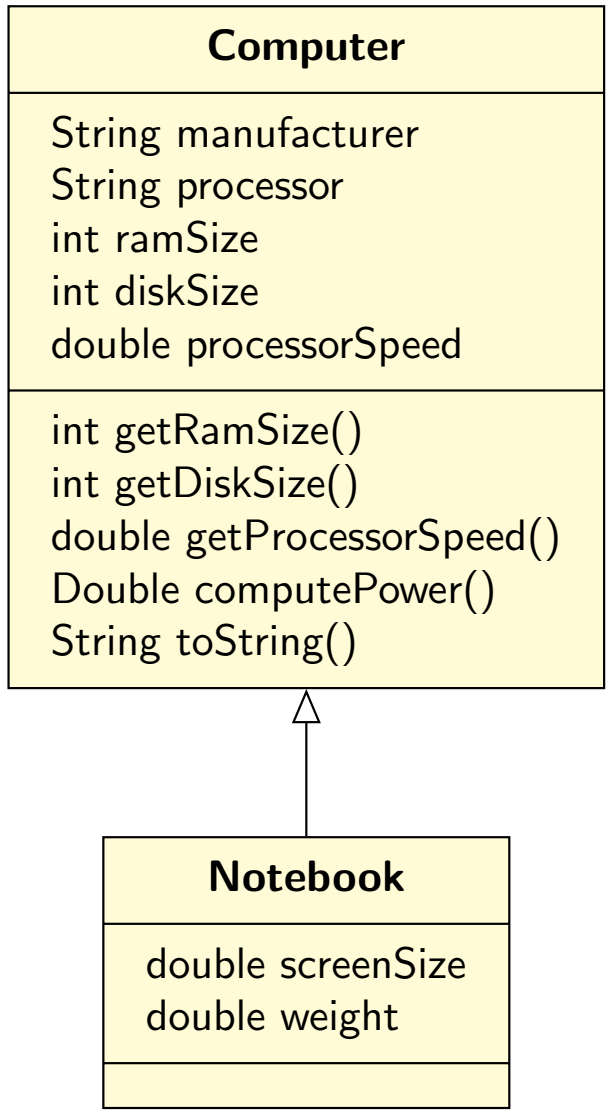
```
Computer myComputer = new Computer(
    "Acme", "Intel", 2, 160, 2.4);
Notebook yourComputer = new Notebook(
    "DellGate", "AMD", 4, 240, 1.8, 15.0, 7.5);
System.out.println(
    "My computer is:\n" + myComputer.toString());
System.out.println(
    "Your computer is:\n" + yourComputer.toString());
```

Method Overriding

This time 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 Overloading

- We now consider method overloading
- Methods with the same name but different parameters are overloaded
- All the overloaded methods are available at the same time

An Example: Overloading Constructors in Notebook

```
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, processor, ram, disk, procSpeed,
  screen, 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 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

A Word on Implicit Casts and Overloading

```
A x;
x=new B();
System.out.print(x.m(5));

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

Output: 20

A Word on Implicit Casts and Overloading

```
A x;
x=new B();
System.out.print(x.m(5));

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

Output: 10

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 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

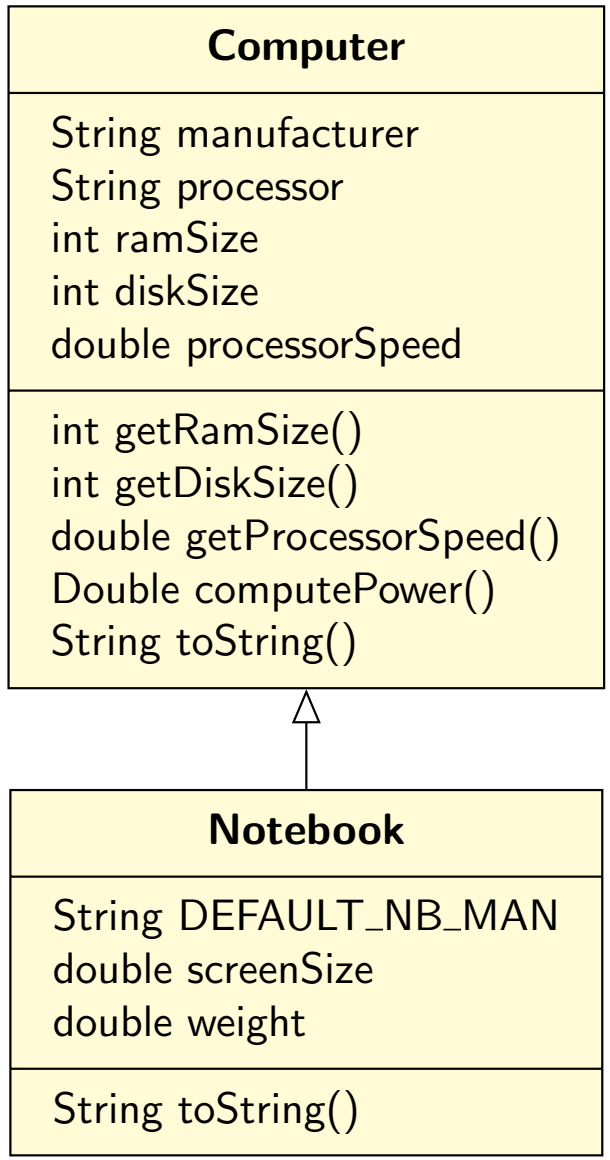
- Suppose the following statements are executed:

```
Computer theComputer = new Notebook(
    "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 Notebook's?

Polymorphism

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



Polymorphism

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

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

Another Example

- 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

Example

- 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 compu
    @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

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

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

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

Abstract Classes

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

```
visibility abstract class className ...
```

- 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

Abstract Classes

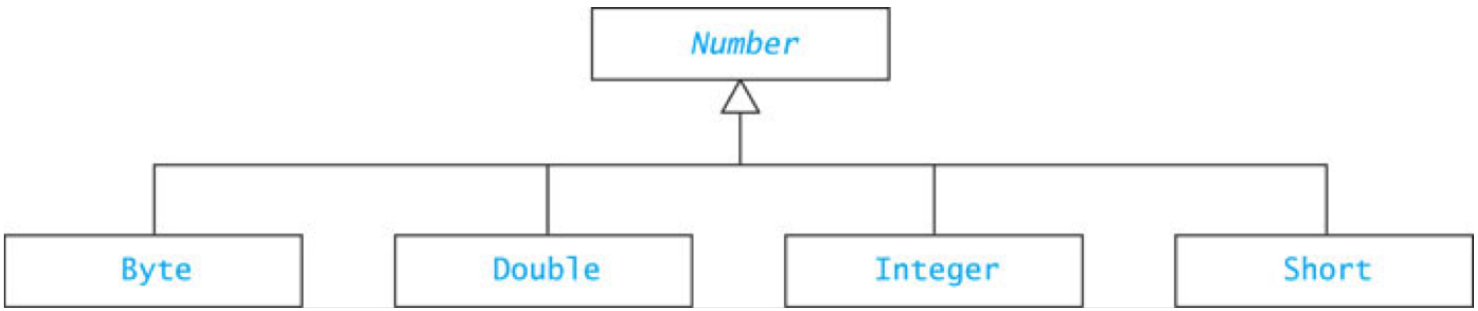
- Use an abstract class in a class hierarchy when you need a base class for two or more subclasses that share some attributes
- You can declare some or all of the attributes and define some or all of the methods that are common to these subclasses
- You can also require that the actual subclasses implement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

Examples of an Abstract Class

```
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();
}
```

Another Example

- 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 and 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 (no body)
  - concrete methods (with a body)
  - data fields
- Abstract classes and Interfaces cannot be instantiated
- Interfaces: allow multiple inheritance, (abstract) classes to not
- Abstract classes: allow code to be shared, interfaces do not

Abstract Classes and 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

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

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

Operations Determined by Type of Reference Variable

- As shown previously with Computer and Notebook, 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)

Operations Determined by Type of Reference Variable (cont.)

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

- The type of the variable determines what operations are legal
- The following is legal: `aThing.toString();`
- But this is not legal: `aThing.intValue();`
- `Object` has a `toString()` method, but it does not have an `intValue()` method (even though `Integer` does, the reference is considered of type `Object`)

Method `Object.equals`

- `Object.equals` method has a parameter of type Object

```
public boolean equals (Object other) {...}
```

- Compares two objects to determine if they are equal
- A class must override equals in order to support comparison

Employee.equals()

```
/** Determines whether the current object matches its argument
    @param obj The object to be compared to the current object
    @return true if the objects have the same name and address
            otherwise, return false
*/
@Override
public boolean equals(Object obj) {
    if (obj == this) return true;
    if (obj == null) return false;
    if (this.getClass() == obj.getClass()) {
        Employee other = (Employee) obj;
        return name.equals(other.name) &&
               address.equals(other.address);
    } else {
        return false;
    }
}
```

Class `Class`

- Every class has a Class object that is created automatically when the class is loaded into an application
- Each Class object is unique for the class
- Method `getClass()` is a member of Object that returns a reference to this unique object
- In the previous example, if

```
this.getClass() == obj.getClass()
```

  is true, then we know that obj and this are both of class Employee

Operations Determined by Type of Reference Variable (cont.)

- The following method will compile,

```
aThing.equals(new Integer("25"));
```

- Object has an equals method, and so does Integer
- Which one is called? Why?
- Why does the following generate a syntax error?

```
Integer aNum = aThing;
```

- Incompatible types!

Casting in a Class Hierarchy

- Casting obtains a reference of a different, but matching, type
- Casting does not change the object! It creates an anonymous reference to the object

```
Integer aNum = (Integer) aThing;
```

- Does this work?

```
((Integer) aThing).intValue()
```

Casting in a Class Hierarchy (cont.)

- Downcast:
  - Cast superclass type to subclass type
  - Java checks at run time to make sure it's legal
  - If it's not legal, it throws ClassCastException
- Upcast:
  - Always valid but unnecessary

Using instanceof to Guard a Casting Operation

instanceof can guard against a `ClassCastException`

```
Object obj = ...;
if (obj instanceof Integer) {
  Integer i = (Integer) obj;
  int val = i;
  ...;
} else {
  ...
}
```

Polymorphism Eliminates Nested if Statements

```
Number[] stuff = new Number[10];
// each element of stuff must reference actual
// object which is a subclass of Number
...

// Non OO style:
if (stuff[i] instanceof Integer)
  sum += ((Integer) stuff[i]).doubleValue();
else if (stuff[i] instanceof Double)
  sum += ((Double) stuff[i]).doubleValue();
...

// OO style:
sum += stuff[i].doubleValue();
```

Polymorphism Eliminates Nested if Statements (cont.)

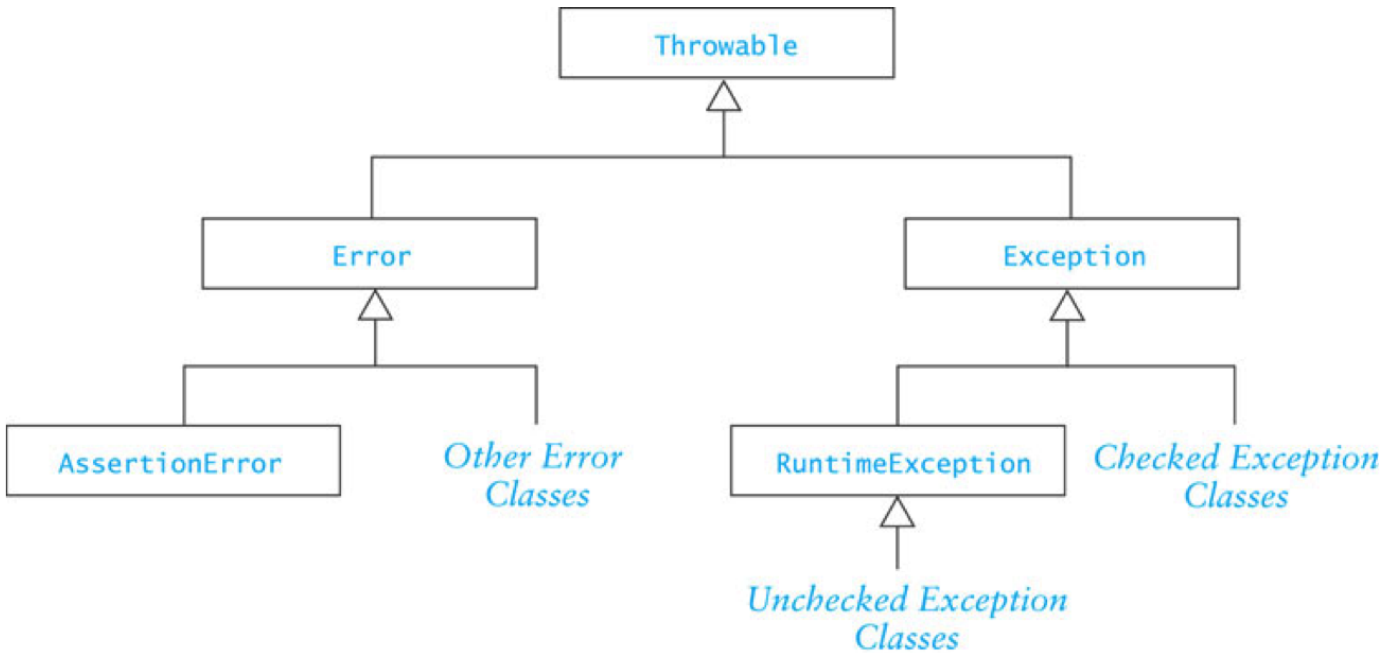
- Polymorphic code style is more extensible; it works automatically with new subclasses
- Polymorphic code is more efficient; the system does one indirect branch versus many tests
- Uses of instanceof may suggest poor coding style

Run-time Errors or Exceptions

- Run-time errors
  - occur during program execution (i.e. at run-time)
  - occur when the JVM detects an operation that it knows to be incorrect
  - cause the JVM to throw an exception
- Examples of run-time errors include
  - division by zero
  - array index out of bounds
  - number format error
  - null pointer exception

Class `Throwable`

- `Throwable` is the superclass of all exceptions
- All exception classes inherit its methods



Checked and Unchecked Exceptions

- Checked exceptions
  - normally not due to programmer error
  - generally beyond the control of the programmer
  - all input/output errors are checked exceptions
  - Examples: IOException, FileNotFoundException
- Unchecked exceptions result from
  - programmer error (try to prevent them with defensive programming)
  - a serious external condition that is unrecoverable
  - Examples: NullPointerException, ArrayIndexOutOfBoundsException

Unchecked Exceptions

- The class `Error` and its subclasses represent errors due to serious external conditions; they are unchecked
  - Example: OutOfMemoryError
  - You cannot foresee or guard against them
  - While you can attempt to handle them, it is generally not a good idea as you will probably be unsuccessful
- The class `Exception` and its subclasses can be handled by a program
  - `RuntimeException` and its subclasses are unchecked
  - All others must be either: explicitly caught or explicitly mentioned as thrown by the method

Checked Example

Suppose we type this code in order to prepare for reading from a text file...

```
File file = new File("file.txt");
BufferedReader reader = new BufferedReader(new FileReader(file));
```

Error: `Unhandled exception type FileNotFoundException`

Some Common Unchecked Exceptions

- `ArithmeticException`: division by zero, etc.
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`: converting a "bad" string to a number
- `NullPointerException`

```
@Override
public boolean equal (Shape s) {
        return this.area()==s.area();
}
```

What if `s` is null? Java does not force us to catch/throw `NullPointerException`

Handling Exceptions

- When an exception is thrown, the normal sequence of execution is interrupted
- Default behavior (no handler)
  - Program stops
  - JVM displays an error message
- The programmer may provide a handle
  - Enclose statements in a `try` block
  - Process the exception in a `catch` block

The `try-catch` Sequence

The try-catch sequence resembles an if-then-else statement

```
try {
  // Execute the following statements until an
  // exception is thrown
  ...
  // Skip the catch blocks if no exceptions were thrown
} catch (ExceptionTypeA ex) {
  // Execute this catch block if an exception of type
  // ExceptionTypeA was thrown in the try block
  ...
} catch (ExceptionTypeB ex) {
  // Execute this catch block if an exception of type
  // ExceptionTypeB was thrown in the try block
  ...
}
```

- `ExceptionTypeB` cannot be a subclass of `ExceptionTypeA`. If is was, its exceptions would be caught be the first catch clause and its catch clause would be unreachable.

Using try-catch

User input is a common source of exceptions

```
public static int getIntValue(Scanner scan) {
  int nextInt = 0;            // next int value
  boolean validInt = false; // flag for valid input
  while(!validInt) {
    try {
      System.out.println("Enter number of kids: ");
      nextInt = scan.nextInt();
      validInt = true;
    } catch (InputMismatchException ex) {
      scan.nextLine();   // clear buffer
      System.out.println("Bad data-enter an integer");
    }
  }
  return nextInt;
}
```

Throwing an Exception When Recovery is Not Obvious

- In some cases, you may be able to write code that detects certain types of errors, but there may not be an obvious way to recover from them
- In these cases an the exception can be thrown
- The calling method receives the thrown exception and must handle it

Throwing an Exception When Recovery is Not Obvious (cont.)

```
public static void processPositiveInteger(int n) {
 if (n < 0) {
   throw new IllegalArgumentException("Invalid argument");
 } else {
   // Process n as required
   ...
 }
}
```

Throwing an Exception When Recovery is Not Obvious (cont.)

A brief side comment: `IllegalArgumentException`, above, is unchecked. The following would not be accepted by Java

```
public static void processPositiveInteger(int n) {
  ... {
   throw new IOException("Invalid");
  }
}
```

We would have to write

```
public static void processPositiveInteger(
         int n) throws IOException {
  ... {
   throw new IOException("Invalid");
  }
}
```

Throwing an Exception When Recovery is Not Obvious (cont.)

```
public static void main(String[] args) {
  Scanner scan = new Scanner(System.in);
  try {
    int num = getIntValue(scan);
    processPositiveInteger(num);
  } catch (IllegalArguementException ex) {
    System.err.println(ex.getMessage());
    System.exit(1);  // error indication
  }
  System.exit(0);  // normal exit
}
```

Method Overriding and Overloading

Polymorphism

Abstract Classes

Class `Object` and Casting

Exceptions

Packages and Visibility

Packages and Visibility

- A Java package is a group of cooperating classes
- The Java API is organized as packages
- Indicate the package of a class at the top of the file:

```
package classPackage;
```

- Classes in the same package should be in the same directory (folder)
- The folder must have the same name as the package
- Classes in the same folder must be in the same package

Packages and Visibility

- Classes not part of a package can only access public members of classes in the package
- If a class is not part of the package, it must access the public classes by their complete name, which would be `packagename.className`
- For example, `x = Java.awt.Color.GREEN;`
- If the package is imported, the packageName prefix is not required.

```
import java.awt.Color;
...
x = Color.GREEN;
```

The Default Package

- Files which do not specify a package are part of the default package
- If you do not declare packages, all of your classes belong to the default package
- The default package is intended for use during the early stages of implementation or for small prototypes
- When you develop an application, declare its classes to be in the same package

Visibility

- We have seen three visibility layers, public, protected, private
- A fourth layer, package visibility, lies between private and protected
- Classes, data fields, and methods with package visibility are accessible to all other methods of the same package, but are not accessible to methods outside the package
- Classes, data fields, and methods that are declared protected are visible within subclasses that are declared outside the package (in addition to being visible to all members inside the package)
- There is no keyword to indicate package visibility
- Package visibility is the default in a package if public, protected, private are not used

Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in Java
  - private: for members that should be invisible even in subclasses
  - package: shields classes and members from classes outside the package
  - protected: provides visibility to extenders or classes in the package
  - public: provides visibility to all
- Encapsulation insulates against change: greater visibility means less encapsulation
- So use the most restrictive visibility possible to get the job done!