

Recapitulate CSE160: Java Classes, Objects, Inheritance, Abstract Classes and Interfaces

CSE260, Computer Science B: Honors

Stony Brook University

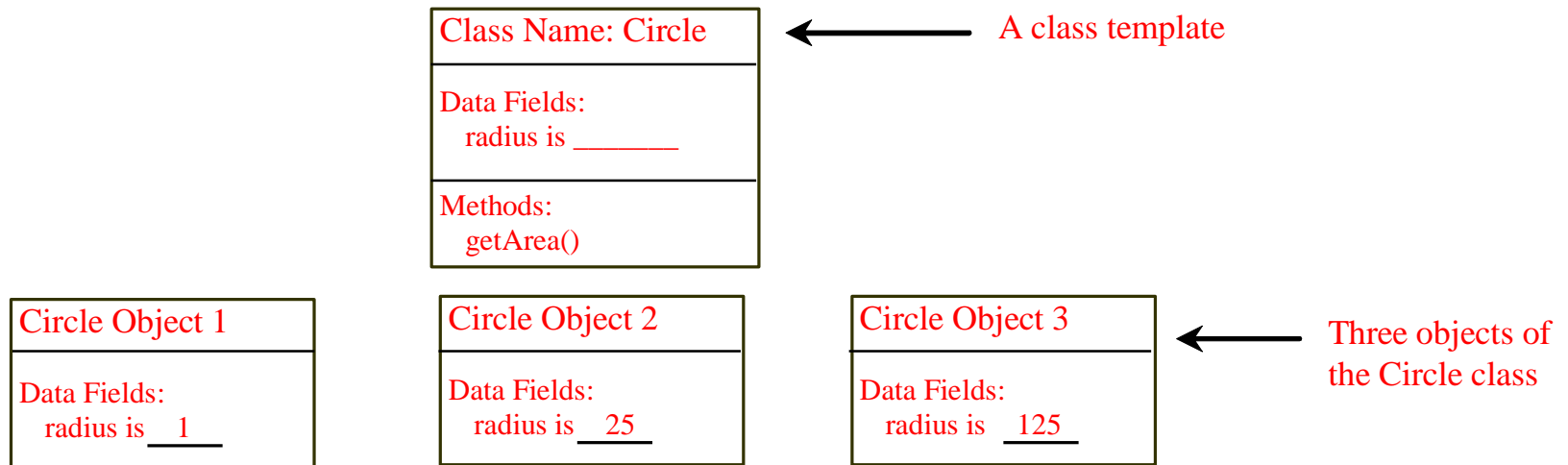
<http://www.cs.stonybrook.edu/~cse260>

Objectives

- To refresh information from CSE160 about classes, objects, inheritance, abstract classes and interfaces

OO Programming Concepts

- An object represents an entity in the real world that can be distinctly identified.
- An object has a unique state and behaviors
 - the state of an object consists of a set of data fields (properties) with their current values
 - the behavior of an object is defined by a set of methods



Classes

- Classes are templates that define objects of the same type.
- A Java class uses:
 - variables to define data fields and
 - methods to define behaviors
- A class provides a special type of methods called **constructors** which are invoked to construct objects from the class

Classes

```
class Circle {  
    /** The radius of this circle */  
    private double radius = 1.0;  
  
    /** Construct a circle object */  
    public Circle() {  
    }  
  
    /** Construct a circle object */  
    public Circle(double newRadius) {  
        radius = newRadius;  
    }  
  
    /** Return the area of this circle */  
    public double getArea()  
        return radius * radius * 3.14159;  
    }  
}
```

← Data field

← Constructors

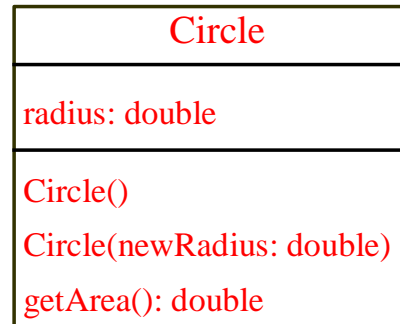
← Method

Classes

```
public class TestCircle {  
  
    public static void main(String[] args) {  
        Circle circle1 = new Circle();  
        Circle circle2 = new Circle(25);  
        Circle circle3 = new Circle(125);  
  
        System.out.println( circle1.getArea() );  
        System.out.println( circle2.getArea() );  
        System.out.println( circle3.getArea() );  
  
        //System.out.println( circle1.radius );  
        //System.out.println( circle2.radius );  
        //System.out.println( circle3.radius );  
    }  
  
}
```

UML Class Diagram

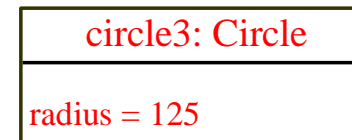
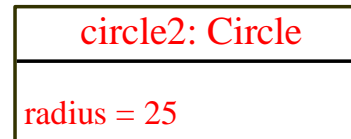
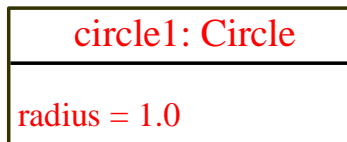
UML Class Diagram



← Class name

← Data fields

← Constructors and methods



← UML notation for objects

Constructors

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the **new** operator when an object is created — they initialize objects to **reference variables**:

```
ClassName o = new ClassName();
```

- Example:

```
Circle myCircle = new Circle(5.0);
```

- A class may be declared without constructors: a no-arg **default constructor** with an empty body is implicitly declared in the class

Accessing Objects

- Referencing the object's data:

`objectRefVar.data`

- Example: **`myCircle.radius`**

- Invoking the object's method:

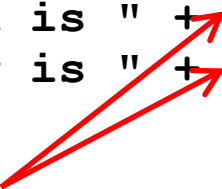
`objectRefVar.methodName(arguments)`

- Example: **`myCircle.getArea()`**

Default values

Java assigns no default value to a local variable inside a method.

```
public class Test {  
    public static void main(String[] args) {  
        int x; // x has no default value  
        String y; // y has no default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```



Compilation error: the variables are not initialized

BUT it assigns default values to data fields!

Reference Data Fields

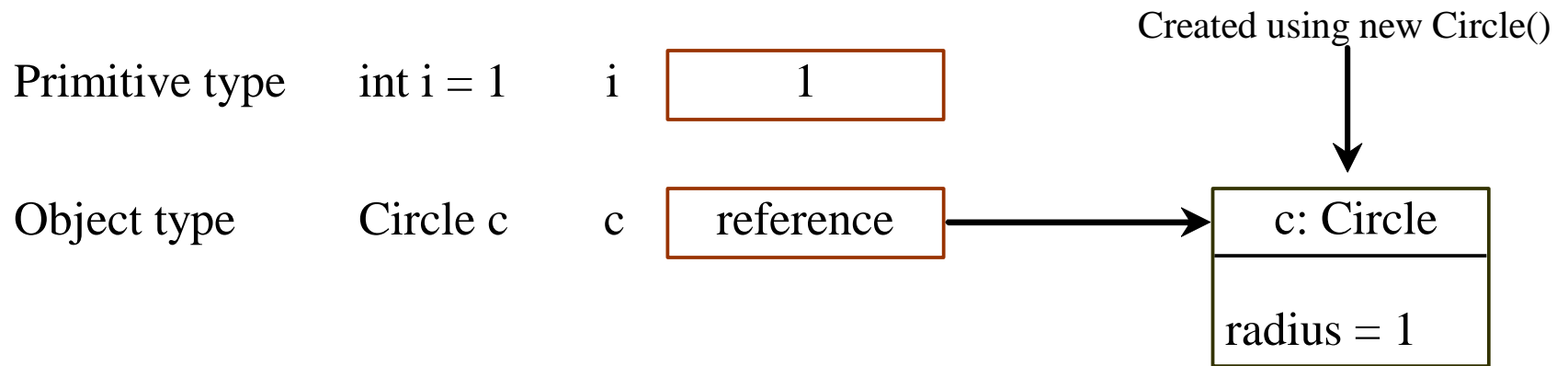
- The data fields can also be of reference types
- Example:

```
public class Student {  
    String name; // name has default value null  
    int age; // age has default value 0  
    boolean isScienceMajor; // isScienceMajor has default value false  
    char gender; // c has default value '\u0000'  
}
```

- If a data field of a reference type does not reference any object, the data field holds a special literal value: **null**.

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name); // null  
        System.out.println("age? " + student.age); // 0  
        System.out.println("isScienceMajor? " + student.isScienceMajor); // false  
        System.out.println("gender? " + student.gender); //  
    }  
}
```

Differences between Variables of Primitive Data Types and Object Types



Copying Variables of Primitive Data Types and Object Types

Primitive type assignment $i = j$

Before:

i 1

j 2

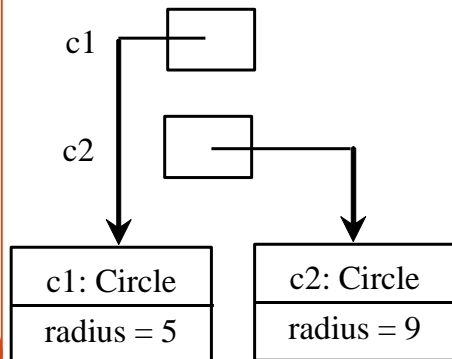
After:

i 2

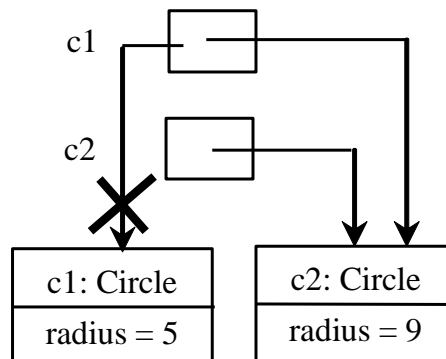
j 2

Object type assignment **c1 = c2**

Before:



After:

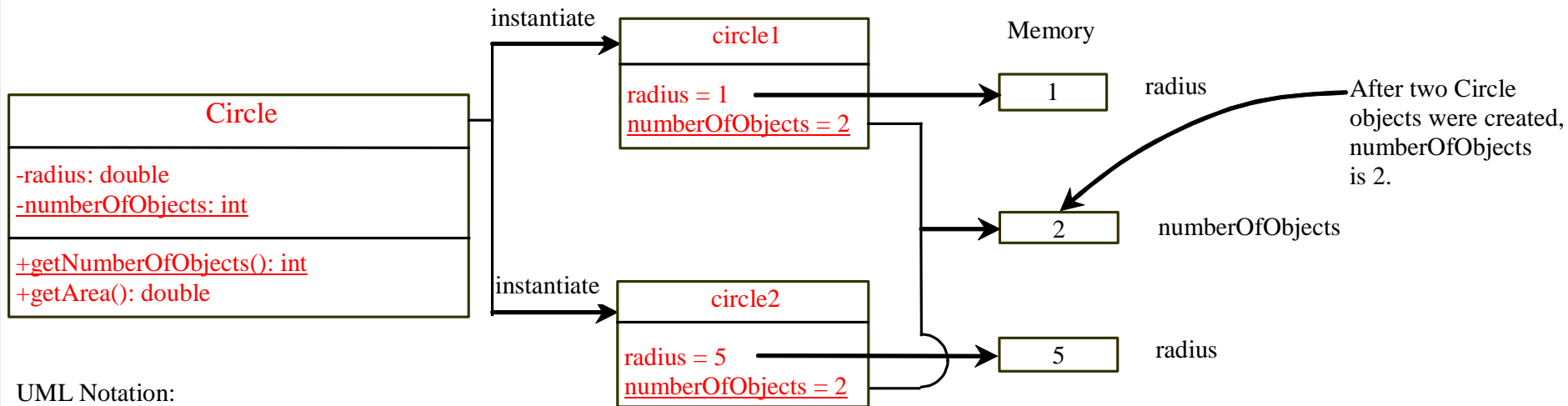


- The object previously referenced by c1 is no longer referenced – it is called *garbage*
- Garbage is automatically collected by JVM = *garbage collection*

static vs. non-static

- Static methods & variables are scoped to a class
 - one static variable for all objects to share!
- Non-static (object) methods & variables are scoped to a single object
 - each object owns its non-static methods & variables

Static Variables, Constants and Methods



UML Notation:

+: public variables or methods

underline: static variables or methods

```
public class StaticExample {  
    public int nonStaticCounter = 0;  
    public static int staticCounter = 0;  
    public StaticExample() {  
        nonStaticCounter++;  
        staticCounter++;  
    }  
    public static void main(String[] args) {  
        StaticExample ex;  
        ex = new StaticExample();  
        ex = new StaticExample();  
        ex = new StaticExample();  
        System.out.println(ex.nonStaticCounter);  
        System.out.println(staticCounter);  
    }  
}
```



```
public class StaticExample {  
    public int nonStaticCounter = 0;  
    public static int staticCounter = 0;  
    public StaticExample() {  
        nonStaticCounter++;  
        staticCounter++;  
    }  
    public static void main(String[] args) {  
        StaticExample ex;  
        ex = new StaticExample();  
        ex = new StaticExample();  
        ex = new StaticExample();  
        System.out.println(ex.nonStaticCounter);  
        System.out.println(staticCounter);  
    }  
}
```

Output: 1

3

static usage

- Can a **static** method:
 - directly call (without using a “.”) a non-**static** method in the same class?
 - directly call a **static** method in the same class?
 - directly reference a non-**static** variable in the same class?
 - directly reference a **static** variable in the same class?
- Can a non-**static** method:
 - directly call (without using a “.”) a non-**static** method in the same class?
 - directly call a **static** method in the same class?
 - directly reference a non-**static** variable in the same class?
 - directly reference a **static** variable in the same class?

static usage

- Can a **static** method:
 - directly call (without using a “.”) a non-**static** method in the same class? **No**
 - directly call a **static** method in the same class? **Yes**
 - directly reference a non-**static** variable in the same class? **No**
 - directly reference a **static** variable in the same class? **Yes**
- Can a non-**static** method:
 - directly call (without using a “.”) a non-**static** method in the same class? **Yes**
 - directly call a **static** method in the same class? **Yes**
 - directly reference a non-**static** variable in the same class? **Yes**
 - directly reference a **static** variable in the same class? **Yes**

```
1 public class Nothing {
2     private int nada;                                // Errors?
3     private static int nothing;
4
5     public void doNada(){ System.out.println(nada);    }
6     public static void doNothing(){ System.out.println("NOTHING"); }
7
8     public static void myStaticMethod()    {
9         doNada();
10        doNothing();
11        nada = 2;
12        nothing = 2;
13        Nothing n = new Nothing();
14        n.doNada();
15        n.nada = 2;
16        n.nothing = 6;
17    }
18    public void myNonStaticMethod() {
19        doNada();
20        doNothing();
21        nada = 2;
22        nothing = 2;
23        Nothing n = new Nothing();
24        n.doNada();
25        n.nada = 2;
26    }}
```

```

1 public class Nothing {
2     private int nada;
3     private static int nothing;
4
5     public void doNada(){ System.out.println(nada);      }
6     public static void doNothing(){ System.out.println("NOTHING"); }
7
8     public static void myStaticMethod()    {
9         doNada();
10        doNothing();
11        nada = 2;
12        nothing = 2;
13        Nothing n = new Nothing();
14        n.doNada();
15        n.nada = 2;
16        n.nothing = 6;
17    }
18    public void myNonStaticMethod() {
19        doNada();
20        doNothing();
21        nada = 2;
22        nothing = 2;
23        Nothing n = new Nothing();
24        n.doNada();
25        n.nada = 2;
26    }}

```

Visibility Modifiers and Accessor/Mutator Methods

- By default, the class, variable, or method can be accessed by any class in the same package.

`public`

The class, data, or method is visible to any class in any package.

`private`

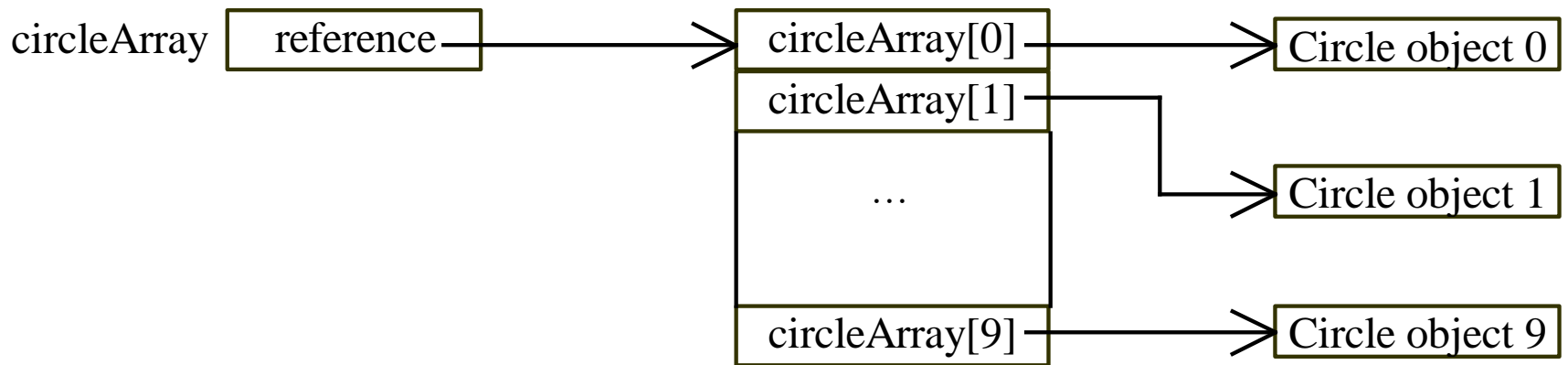
The data or methods can be accessed only by the declaring class - To protect data!

The get and set methods are used to read and modify private properties.

Array of Objects

```
Circle[] circleArray = new Circle[10];
```

- An *array of objects* is an *array of reference variables* (like the multi-dimensional arrays seen before)



Immutable Objects and Classes

- Immutable object: the contents of an object cannot be changed once the object is created - its class is called an immutable class
- Example immutable class: no set method in the Circle class

```
public class Circle{  
    private double radius;  
    public Circle() { }  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    public double getRadius() {  
        return radius;  
    }  
}
```

- radius is private and cannot be changed without a set method
- A class with all private data fields and without mutators is not necessarily immutable!

What Class is Immutable?

1. It must mark all data fields private!
2. Provide no mutator methods!
3. Provide no accessor methods that would return a reference to a mutable data field object!

Example mutable

```
public class Student {
    private int id;
    private BirthDate birthDate;
    public Student(int ssn, int year,
        int month, int day) {
        id = ssn;
        birthDate =
            new BirthDate(year, month, day);
    }
    public int getId() {
        return id;
    }
    public BirthDate getBirthDate() {
        return birthDate;
    }
}
```

```
public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int newYear,
        int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }
    public void setYear(int newYear) {
        year = newYear;
    }
    public int getYear() {
        return year;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1998, 1, 1);
        BirthDate date = student.getBirthDate();
        date.setYear(2050);
        // Now the student birth year is changed:
        System.out.println(student.getBirthDate().getYear()); // 2050
    }
}
```

The **this** Keyword

- The **this** keyword is the name of a reference that refers to an object itself
- Common uses of the **this** keyword:
 1. Reference a class's “*hidden*” *data fields*
 2. To enable a constructor to invoke another constructor of the same class as the first statement in the constructor.

Reference the Hidden Data Fields

```
public class Foo {  
    private int i = 5;  
    private static double k = 0;  
  
    void setI(int i) {  
        this.i = i;  
    }  
  
    static void setK(double k) {  
        Foo.k = k;  
    }  
}
```

Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
 this.i = 10, where **this** refers f1

Invoking f2.setI(45) is to execute
 this.i = 45, where **this** refers f2

Calling Overloaded Constructor

```
public class Circle {  
    private double radius;
```

```
    public Circle(double radius) {  
        this.radius = radius;  
    }
```

→ this must be explicitly used to reference the data field radius of the object being constructed

```
    public Circle() {  
        this(1.0);  
    }
```

→ this is used to invoke another constructor

```
    public double getArea() {  
        return this.radius * this.radius * Math.PI;  
    }
```

↓
Every instance variable belongs to an instance represented by this, which is normally omitted

The Date Class

Java provides a system-independent encapsulation of date and time in the java.util.Date class.

The toString method returns the date and time as a string

The + sign indicates
public modifier



java.util.Date	
+Date()	
+Date(elapseTime: long)	
+toString(): String	
+getTime(): long	
+setTime(elapseTime: long): void	

Constructs a Date object for the current time.

Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.

Returns a string representing the date and time.

Returns the number of milliseconds since January 1, 1970, GMT.

Sets a new elapse time in the object.

January 1, 1970, GMT is called
the Unix time (or Unix epoch time)

```
java.util.Date date = new java.util.Date();  
System.out.println(date.toString());
```

Constructing Strings

- Pattern:

```
String newString = new String(stringLiteral);
```

- Example:

```
String message = new String("Welcome to Java");
```

- Since strings are used frequently, Java provides a shorthand initializer for creating a string:

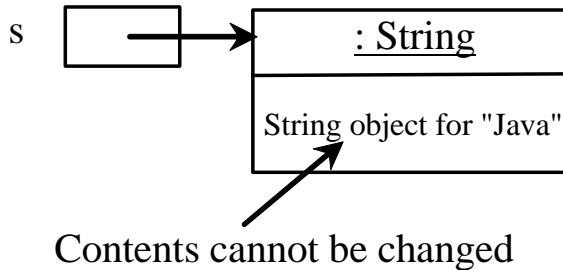
```
String message = "Welcome to Java";
```

Strings Are Immutable

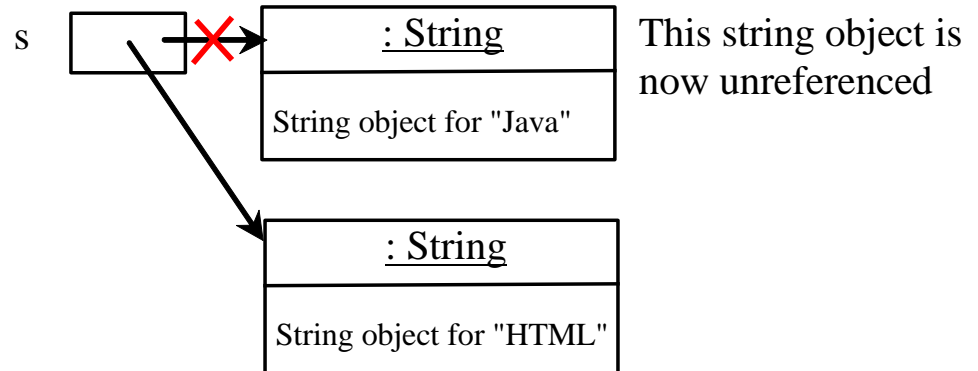
- A String object is immutable; its contents cannot be changed

```
String s = "Java";  
s = "HTML";
```

After executing `String s = "Java";`



After executing `s = "HTML";`



Interned Strings

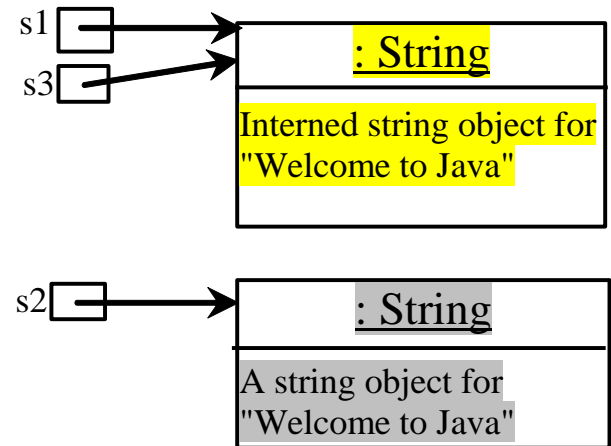
```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

```
String s3 = "Welcome to Java";
```

```
System.out.println("s1 == s2 is " + (s1 == s2));
```

```
System.out.println("s1 == s3 is " + (s1 == s3));
```



display

s1 == s2 is false

s1 == s3 is true

- A new object is created if you use the new operator.
- If you use the string initializer, no new object is created if the interned object is already created.

Matching, Replacing and Splitting by Patterns

- You can match, replace, or split a string by **specifying a pattern**
= *regular expressions*

```
"Java is fun".matches("Java.*") ;
```

```
"Java is cool".matches("Java.*") ;
```

Social security numbers is **xxx-xx-xxxx**, where **x** is a digit:

```
[\\d]{3}-[\\d]{2}-[\\d]{4}
```

An even number ends with digits **0, 2, 4, 6, or 8**:

```
[\\d]*[02468]
```

Telephone numbers **(xxx) xxx-xxxx**, where **x** is a digit and the first digit cannot be zero:

```
\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}
```

Regular Expressions

Regular Expression	Matches	Example
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab cd)	ab or cd	ten matches t(en im)
[abc]	a, b, or c	Java matches Ja[uvw x]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [1-9]	Java2 matches "Java[\\d]"
\D	a non-digit	\$Java matches "[\\D][\\D]ava"
\w	a word character	Java matches "[\\w]ava"
\W	a non-word character	\$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
p*	zero or more occurrences of pattern p	Java matches "[\\w]*"
p+	one or more occurrences of pattern p	Java matches "[\\w]+"
p?	zero or one occurrence of pattern p	Java matches "[\\w]?Java"
p{n}	exactly n occurrences of pattern p	Java matches "[\\w]?ava"
p{n,}	at least n occurrences of pattern p	Java matches "[\\w]{4}"
p{n,m}	between n and m occurrences (inclusive)	Java matches "[\\w]{3,}"
		Java matches "[\\w]{1,9}"

Command-Line Parameters

```
class TestMain {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

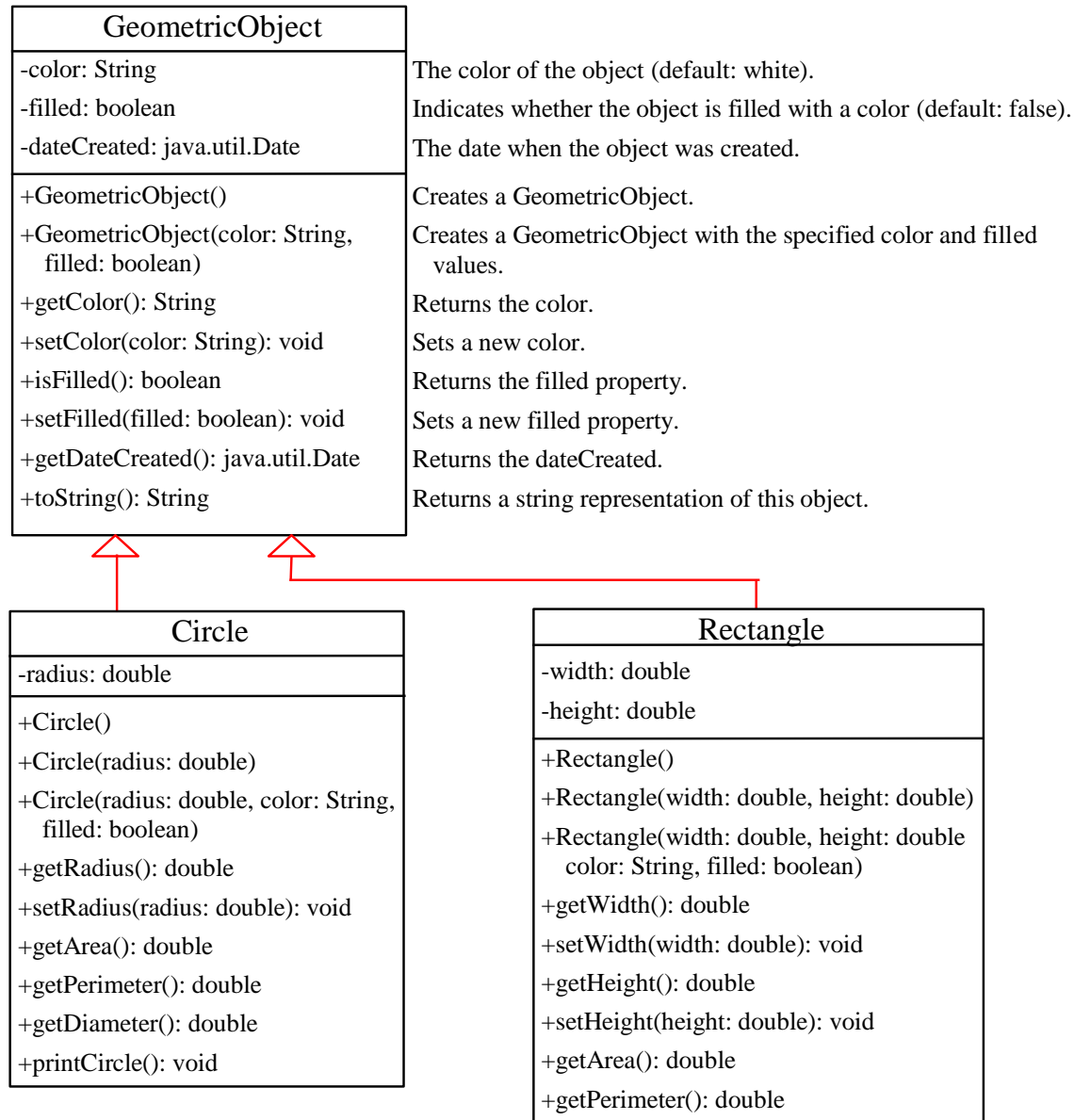
- Run with:

`java TestMain arg0 arg1 arg2 ... argn`
or EclipseIDE arguments

StringBuilder and StringBuffer

- The StringBuilder/StringBuffer class is an alternative to the String class:
 - StringBuilder/StringBuffer can be used wherever a string is used
 - StringBuilder/StringBuffer is more flexible than String
 - You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created

Inheritance



```

public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;
    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }
    protected GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }
    public String getColor() {    return color;    }
    public void setColor(String color) {    this.color = color;    }
    public boolean isFilled() {    return filled;    }
    public void setFilled(boolean filled) {    this.filled = filled;    }
    public java.util.Date getDateCreated() {    return dateCreated;    }
    public String toString() {
        return "created on " + dateCreated + "\n" + "color: " + color +
            " and filled: " + filled;
    }
    /** Abstract method getArea */
    public abstract double getArea();
    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}

```

```
public class Circle extends GeometricObject {
    private double radius;
    public Circle() { }
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
    public double getDiameter() {
        return 2 * radius;
    }
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    /* Print the circle info */
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() +
            " and the radius is " + radius);
    }
}
```



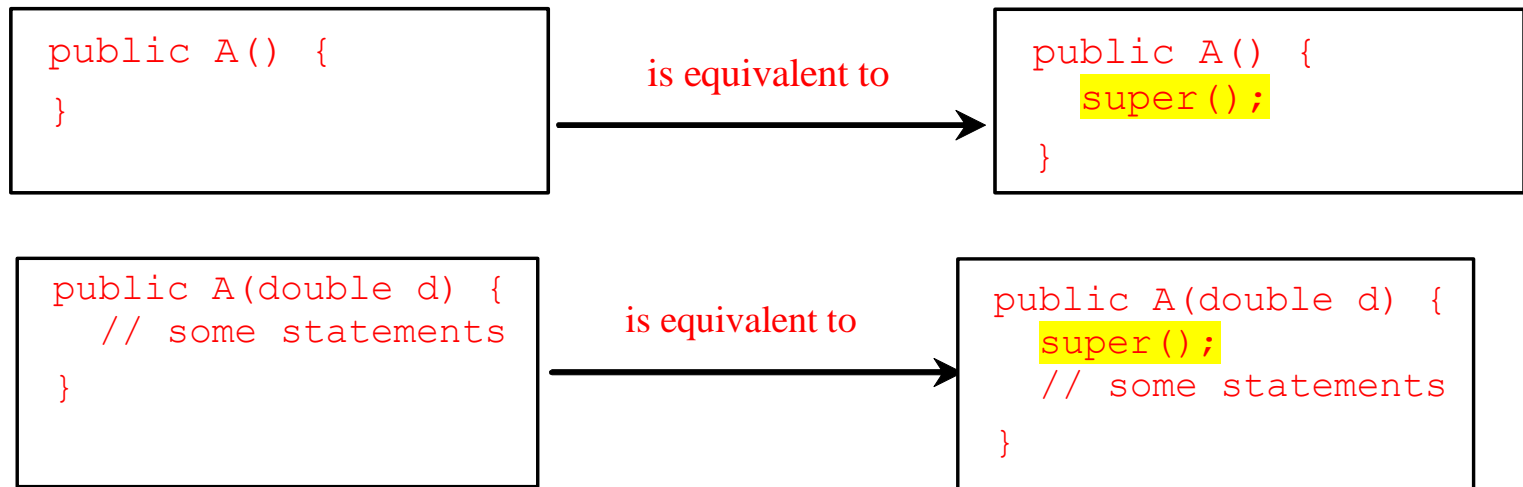
```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle() { }
    public Rectangle(double width, double height, String color,
        boolean filled) {
        super(color,filled);
        this.width = width;
        this.height = height;
    }
    public double getWidth() {    return width;    }
    public void setWidth(double width) {        this.width = width;    }
    public double getHeight() {    return height;    }
    public void setHeight(double height) {        this.height = height;    }

    public double getArea() {
        return width * height;
    }

    public double getPerimeter() {
        return 2 * (width + height);
    }
}
```

Are superclass's Constructor Inherited?

- No. They are not inherited.
- They are invoked explicitly or implicitly:
 - Explicitly using the **super** keyword
 - If the keyword **super** is not explicitly used, the superclass's no-arg constructor is automatically invoked as the first statement in the constructor, unless another constructor is invoked (when the last constructor in the chain will invoke the superclass constructor)



Using the Keyword **super**

- The keyword **super** refers to the superclass of the class in which **super** appears. This keyword can be used in two ways:
 - To call a superclass constructor: Java requires that the statement that uses the keyword **super** appear first in the constructor (unless another constructor is called or the superclass constructor is called implicitly)
 - To call a superclass method

Constructor Chaining

- *Constructor chaining* : constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.

```
public class Faculty extends Employee {
    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
    public static void main(String[] args) {
        new Faculty();
    }
}
class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }
    public Employee(String s) {
        System.out.println(s);
    }
}
class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

1. Start from the
main method

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Invoke Faculty
constructor

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

3. Invoke Employee's no-arg constructor

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
}
```

```
public Faculty() {  
    System.out.println("(4) Faculty's no-arg constructor is invoked");  
}  
}
```

4. Invoke Employee(String)
constructor

```
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
}
```

```
public Employee(String s) {  
    System.out.println(s);  
}  
}
```

```
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```


Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

5. Invoke Person() constructor

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

6. Execute println

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

7. Execute println

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

8. Execute println

Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

9. Execute println

Calling Superclass Methods

```
public void printCircle() {  
    System.out.println(  
        "The circle is created " +  
        super.getDateCreated() +  
        " and the radius is " +  
        radius);  
}
```

Declaring a Subclass

- A subclass extends properties and methods from the superclass.
- You can also:
 - Add new properties
 - Add new methods
 - Override the methods of the superclass

Overriding Methods in the Superclass

- *Method overriding*: modify in the subclass the implementation of a method defined in the superclass:

```
public class Circle extends GeometricObject {  
    /** Override the toString method defined in  
        GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
    // Other methods are omitted  
    ...  
}
```


Overriding Methods in the Superclass

- An instance method can be overridden only if it is accessible
 - A private method cannot be overridden, because it is not accessible outside its own class
 - If a method **defined in a subclass** is **private in its superclass**, the **two methods** are completely **unrelated**
- A static method can be inherited
 - A static method cannot be overridden
 - If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden

Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);           10.0  
    }                       10.0  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);           10  
    }                       20.0  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

The Object Class and Its Methods

- Every class in Java is descended from the **java.lang.Object** class
- If no inheritance is specified when a class is defined, the superclass of the class is **java.lang.Object**

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```

The toString() method in Object

- The toString() method returns a string representation of the object
- The default Object implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object

```
Loan loan = new Loan();
```

```
System.out.println(loan.toString());
```

- The code displays something like Loan@12345e6
 - you should override the toString() method so that it returns an informative string representation of the object

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {  
    public static void main(String[] args) {  
        m(new GraduateStudent());  
        m(new Student());  
        m(new Person());  
        m(new Object());  
    }  
    public static void m(Object x) {  
        System.out.println(x.toString());  
    }  
}  
class GraduateStudent  
    extends Student {  
}  
class Student extends Person {  
    public String toString() {  
        return "Student";  
    }  
}  
class Person extends Object {  
    public String toString() {  
        return "Person";  
    }  
}
```

Method `m` takes a parameter of the `Object` type – can be invoked with any object

Polymorphism: an object of a subtype can be used wherever its supertype value is required

Dynamic binding: the Java Virtual Machine determines dynamically at runtime which implementation is used by the method

When the method `m(Object x)` is executed, the argument `x`'s `toString` method is invoked.

Output:

Student

Student

Person

java.lang.Object@12345678

Dynamic Binding

- Suppose an object o is an instance of classes C_1, C_2, \dots, C_{n-1} , and C_n
 - C_1 is a subclass of C_2 , C_2 is a subclass of C_3 , ..., and C_{n-1} is a subclass of C_n
 - C_n is the most general class, and C_1 is the most specific class
 - If o invokes a method p , the JVM searches the implementation for the method p in C_1, C_2, \dots, C_{n-1} and C_n , in this order, until it is found, the search stops and the first-found implementation is invoked



Since o is an instance of C_1 , o is also an instance of C_2, C_3, \dots, C_{n-1} , and C_n

Dynamic Binding

```
public class PolymorphismDemo {  
    public static void main(String[] args) {  
        m(new GraduateStudent());  
        m(new Student());  
        m(new Person());  
        m(new Object());  
    }  
    public static void m(Object x) {  
        System.out.println(x.toString());  
    }  
}  
class GraduateStudent extends Student {  
}  
class Student extends Person {  
    public String toString() {  
        return "Student";  
    }  
}  
class Person extends Object {  
    public String toString() {  
        return "Person";  
    }  
}
```

Output:

Student

Student

Person

java.lang.Object@12345678

Method Matching vs. Binding

- The compiler **finds a matching method** according to parameter type, number of parameters, and order of the parameters **at compilation time**
- The Java Virtual Machine **dynamically binds the implementation of the method at runtime**

Casting Objects

- Casting can be used to convert an object of one class type to another within an inheritance hierarchy

```
m(new Student()) ;
```

is equivalent to:

```
Object o = new Student() ;    // Implicit casting  
m(o) ;
```



Legal because an instance of Student is automatically an instance of Object

Student extends Person

```
public class Person {
    public String firstName;
    public String lastName;
    public String toString(){
        return firstName + " " + lastName;
    }
}

public class Student extends Person {
    public double GPA;
    public String toString(){
        return "Student: " + super.toString()
            + ", gpe: " + GPA;
    }
}
```

Person.java

Student.java

Class Casting

- An object can be cast to an ancestor type
 - Which lines would produce compiler errors?
 - Which lines would produce run-time errors?

```
Person p = new Person();  
Student s = new Student();  
p = new Student();  
s = new Person();  
p = (Person)new Student();  
p = (Student)new Student();  
s = (Person)new Person();  
s = (Student)new Person();
```

Class Casting

- An object can be cast to an ancestor type
 - Which lines would produce compiler errors?
 - Which lines would produce run-time errors?

```
Person p = new Person();  
Student s = new Student();  
p = new Student();  
s = new Person();  
p = (Person)new Student();  
p = (Student)new Student();  
s = (Person)new Person();  
s = (Student)new Person();
```

Objects as Boxes of Data

- When you call **new**, you get an id (reference or address) of a box
 - you can give the address to variables
 - variables can share the same address
 - after **new**, we can't add variables/properties to the box
- These rules explain why implicit casting is legal:

```
Person p = new Student();
```

firstName:	null
lastName:	null
GPA:	0.0

- But this is not:

```
Student s = new Person();
```

firstName:	null
lastName:	null

Apparent vs. Actual

- In Java, objects have 2 types:
 - **Apparent** type
 - the type an object variable was **declared** as
 - the **compiler** only cares about this type
 - **Actual** type
 - the type an object variable was **constructed** as
 - the **JVM** only cares about this type

Example: **Person** p = new **Student**(...);

- Very important for method arguments and returned objects

Apparent vs. Actual

```
public class ActualVsApparentExample {  
    public static void main(String[] args){  
        Person p = new Person();  
        p.firstName = "Joe";  
        p.lastName = "Shmo";  
        print(p);  
        p = new Student();  
        p.firstName = "Jane";  
        p.lastName = "Doe";  
        print(p);  
        Student s = (Student)p;  
        print(s);  
    }  
    public static void print(Person p){  
        System.out.println(p);  
    }  
}
```

ActualVsApparentExample.java

Apparent vs. Actual

- Apparent data type of an object determines what methods may be called
- Actual data type determines where the implementation of a called method is defined
 - JVM look first in actual type class & works its way up
 - Dynamic binding

Why Casting Is Necessary?

Student b = o;

- A **compilation error** would occur because an **Object** **o** is not necessarily an instance of **Student**
- We use **explicit casting** to tell the compiler that **o** is a **Student** object - syntax is similar to the one used for casting among primitive data types

Student b = (Student) o;

- This type of casting may not always succeed at run-time (we can check this with **instanceof** operator)

The `instanceof` Operator

- Use the **`instanceof`** operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();  
...  
if (myObject instanceof Circle) {  
    System.out.println("The circle diameter is "  
        + ((Circle)myObject).getDiameter());  
    ...  
}
```

```

public class CastingDemo{
    public static void main(String[] args){
        Object object1 = new Circle(1);
        Object object2 = new Rectangle(1, 1);
        displayObject(object1);
        displayObject(object2);
    }
    public static void displayObject(Object object) {
        if (object instanceof Circle) {
            System.out.println("The circle area is " +
                               ((Circle)object).getArea());
            System.out.println("The circle diameter is " +
                               ((Circle)object).getDiameter());
        }else if (object instanceof Rectangle) {
            System.out.println("The rectangle area is " +
                               ((Rectangle)object).getArea());
        }
    }
}

```

The `equals` Method

- The **`equals()`** method compares the **contents** of two objects - the default implementation of the **`equals`** method in the `Object` class is as follows:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

- **Override the `equals()` method** in other classes:

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else return false;  
}
```

Call-by-Value for objects

- Java methods always use call-by-value:
 - method arguments are *copied* when sent
 - this includes object **ids**

Call-by-Value

CallByValueTester1.java

```
public class CallByValueTester1 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.firstName = "Joe";  
        foo(p);  
        System.out.println(p.firstName);  
    }  
    public static void foo(Person fooPerson) {  
        fooPerson = new Person();  
        fooPerson.firstName = "Bob";  
    }  
}
```

Call-by-Value

```
public class CallByValueTester1 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.firstName = "Joe";  
        foo(p);  
        System.out.println(p.firstName);  
    }  
    public static void foo(Person fooPerson) {  
        fooPerson = new Person();  
        fooPerson.firstName = "Bob";  
    }  
}
```

Output: Joe

Call-by-Value

CallByValueTester2.java

```
public class CallByValueTester2 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.firstName = "Joe";  
        foo(p);  
        System.out.println(p.firstName);  
    }  
    public static void foo(Person fooPerson) {  
        fooPerson.firstName = "Bob";  
        fooPerson = new Person();  
        fooPerson.firstName = "Chris";  
    }  
}
```


Call-by-Value

```
public class CallByValueTester2 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.firstName = "Joe";  
        foo(p);  
        System.out.println(p.firstName);  
    }  
    public static void foo(Person fooPerson) {  
        fooPerson.firstName = "Bob";  
        fooPerson = new Person();  
        fooPerson.firstName = "Chris";  
    }  
}
```

Output: Bob

Call-by-Value

CallByValueTester3.java

```
public class CallByValueTester3 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.firstName = "Joe";  
        p = foo(p);  
        System.out.println(p.firstName);  
    }  
    public static Person foo(Person fooPerson) {  
        fooPerson.firstName = "Bob";  
        fooPerson = new Person();  
        fooPerson.firstName = "Chris";  
        return fooPerson;  
    }  
}
```

Call-by-Value

```
public class CallByValueTester3 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.firstName = "Joe";  
        p = foo(p);  
        System.out.println(p.firstName);  
    }  
    public static Person foo(Person fooPerson) {  
        fooPerson.firstName = "Bob";  
        fooPerson = new Person();  
        fooPerson.firstName = "Chris";  
        return fooPerson;  
    }  
}
```

The ArrayList Class

You can create arrays to store objects - But the array's size is fixed once the array is created. **Java provides the java.util.ArrayList** class that can be used to store an unlimited number of objects:

java.util.ArrayList	
+ArrayList()	Creates an empty list.
+add(o: Object) : void	Appends a new element o at the end of this list.
+add(index: int, o: Object) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : Object	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : Object	Removes the element at the specified index.
+set(index: int, o: Object) : Object	Sets the element at the specified index.

```

public class TestArrayList {
    public static void main(String[] args) {          // Warnings
        java.util.ArrayList cityList = new java.util.ArrayList();
        cityList.add("London");cityList.add("New York");cityList.add("Paris");
        cityList.add("Toronto");cityList.add("Hong Kong");
        System.out.println("List size? " + cityList.size());
        System.out.println("Is Toronto in the list? " +
                           cityList.contains("Toronto"));
        System.out.println("The location of New York in the list? " +
                           cityList.indexOf("New York"));
        System.out.println("Is the list empty? " + cityList.isEmpty()); // false
        cityList.add(2, "Beijing");
        cityList.remove("Toronto");
        for (int i = 0; i < cityList.size(); i++)
            System.out.print(cityList.get(i) + " ");
        System.out.println();
        // Create a list to store two circles
        java.util.ArrayList list = new java.util.ArrayList();
        list.add(new Circle(2));
        list.add(new Circle(3));
        System.out.println( ((Circle) (list.get(0))).getArea() );
    }
}

```

```

public class TestArrayList {
    public static void main(String[] args) {
        java.util.ArrayList<String> cityList=new java.util.ArrayList<String>();
        cityList.add("London");cityList.add("New York");cityList.add("Paris");
        cityList.add("Toronto");cityList.add("Hong Kong");
        System.out.println("List size? " + cityList.size());
        System.out.println("Is Toronto in the list? " +
                           cityList.contains("Toronto"));
        System.out.println("The location of New York in the list? " +
                           cityList.indexOf("New York"));
        System.out.println("Is the list empty? " + cityList.isEmpty()); // false
        cityList.add(2, "Beijing");
        cityList.remove("Toronto");
        for (int i = 0; i < cityList.size(); i++)
            System.out.print(cityList.get(i) + " ");
        System.out.println();
        // Create a list to store two circles
        java.util.ArrayList<Circle> list = new java.util.ArrayList<Circle>();
        list.add(new Circle(2));
        list.add(new Circle(3));
        System.out.println( list.get(0).getArea() );
    }
}

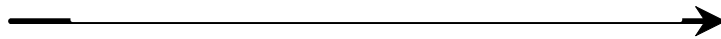
```

// Generics: eliminates warnings

The **protected** Modifier

- A **protected** data or a protected method in a public class **can be accessed by any class in the same package or its subclasses**, even if the subclasses are in a different package

Visibility increases



private, default (if no modifier is used), protected, public

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers

package p1;

```
public class C1 {  
    public int x;  
    protected int y;  
    int z;  
    private int u;  
  
    protected void m() {  
    }  
}
```

```
public class C2 {  
    C1 o = new C1();  
    can access o.x;  
    can access o.y;  
    can access o.z;  
    cannot access o.u;  
  
    can invoke o.m();  
}
```



```
public class C3  
    extends C1 {  
    can access x;  
    can access y;  
    can access z;  
    cannot access u;  
  
    can invoke m();  
}
```

package p2;

```
public class C4  
    extends C1 {  
    can access x;  
    can access y;  
    cannot access z;  
    cannot access u;  
  
    can invoke m();  
}
```

```
public class C5 {  
    C1 o = new C1();  
    can access o.x;  
    cannot access o.y;  
    cannot access o.z;  
    cannot access o.u;  
  
    cannot invoke o.m();  
}
```

The **final** Modifier

- A **final** variable is a constant:

```
final static double PI = 3.14159;
```

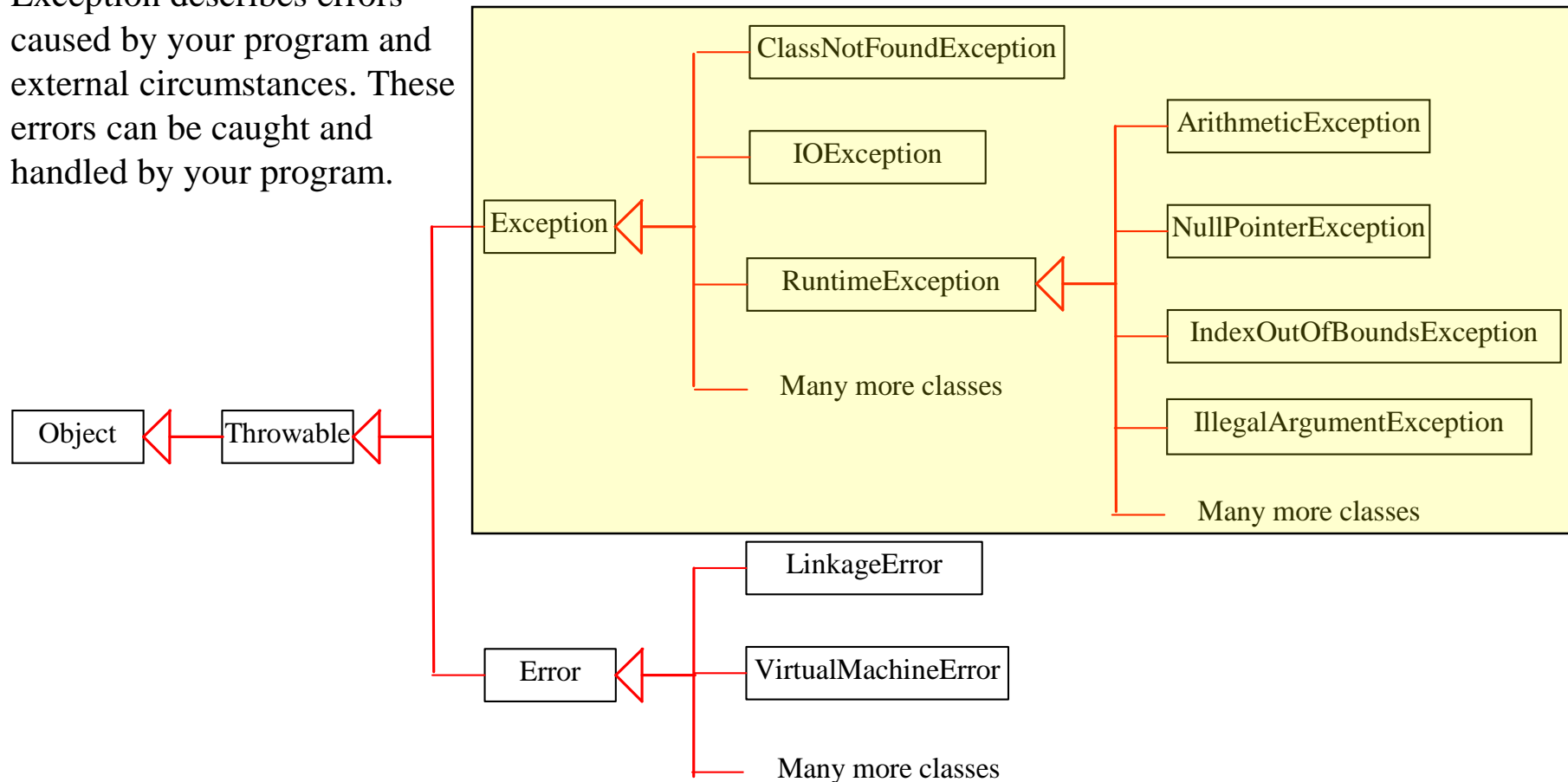
- A **final** method cannot be overridden by its subclasses

- A **final** class cannot be extended:

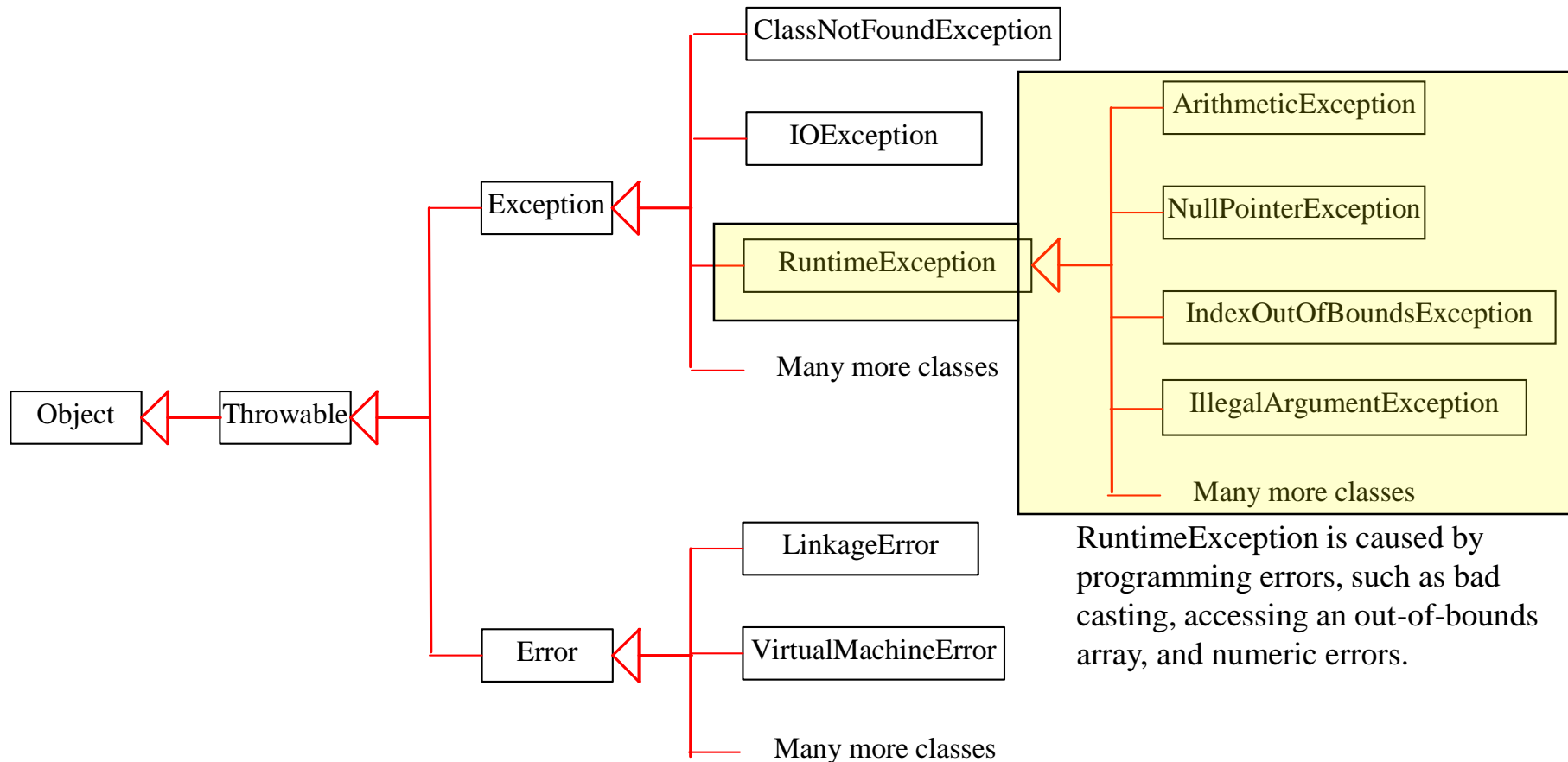
```
final class Math {  
    ...  
}
```

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions



Checked Exceptions vs. Unchecked Exceptions

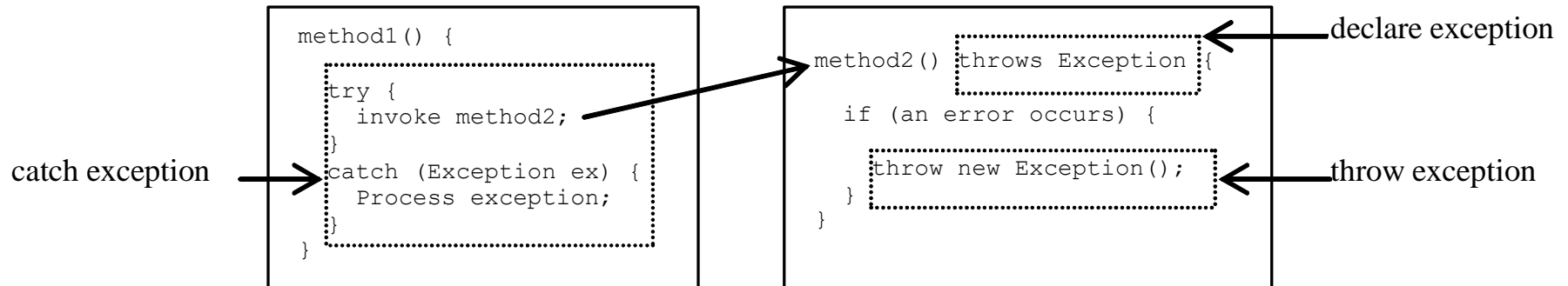
RuntimeException, Error and their subclasses are known as *unchecked exceptions*.

All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions

- In most cases, **unchecked exceptions reflect programming logic errors that are not recoverable.**
 - For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array.
 - **These are the logic errors that should be corrected in the program.**
 - Unchecked exceptions can occur anywhere in the program.
 - To avoid cumbersome overuse of try-catch blocks, **Java does not mandate you to write code to catch unchecked exceptions.**

Declaring, Throwing, and Catching Exceptions



Catch or Declare Checked Exceptions

- Java forces you to deal with **checked exceptions**:
 - If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method
 - For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code: (a) or (b):

```
void p1() {  
    try {  
        p2();  
    }  
    catch (IOException ex) {  
        ...  
    }  
}
```

(a)

```
void p1() throws IOException {  
    p2();  
}
```

(b)


```

public class CircleWithException {
    private double radius; /** The radius of the circle */
    private static int numberOfObjects = 0; /** The number of the objects created
    */
    public CircleWithException() {this(1.0); } /** Construct a circle with radius 1
    */
    public CircleWithException(double newRadius) throws IllegalArgumentException {
        setRadius(newRadius);
        numberOfObjects++;
    }
    public double getRadius() { return radius; }
    public void setRadius(double newRadius) throws IllegalArgumentException {
        if (newRadius >= 0)
            radius = newRadius;
        else
            throw new IllegalArgumentException("Radius cannot be negative");
    }
    public static int getNumberOfObjects() {/** Return numberOfObjects */
        return numberOfObjects;
    }
    public double findArea() {/** Return the area of this circle */
        return radius * radius * 3.14159;
    }
}

```

```
public static void main(String[] args) {  
    try {  
        CircleWithException c1 = new CircleWithException(5);  
        CircleWithException c2 = new CircleWithException(-5);  
        CircleWithException c3 = new CircleWithException(10);  
    } catch (IllegalArgumentException ex) {  
        System.out.println(ex);  
    }  
    System.out.println("Number of objects created: " +  
        CircleWithException.getNumberOfObjects());  
}  
}
```

Output:
Radius cannot be negative
Number of objects created: 1

The `finally` Clause

```
try {  
    statements;  
} catch (TheException ex) {  
    handling ex;  
} finally {  
    finalStatements;  
}
```

The `finally` block *always* executes when the `try` block exits
Useful for cleanup code:

```
}finally {  
    if (out != null) {  
        System.out.println("Closing PrintWriter");  
        out.close();  
    }  
}
```

Custom Exception Class Example

```
public class InvalidRadiusException extends Exception {  
    private double radius;  
  
    /** Construct an exception */  
    public InvalidRadiusException(double radius) {  
        super("Invalid radius " + radius);  
        this.radius = radius;  
    }  
  
    /** Return the radius */  
    public double getRadius() {  
        return radius;  
    }  
}
```

Text I/O: The **File** Class

- The **File** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
 - The filename is a string
 - The **File** class is a wrapper class for the file name and its directory path

Obtaining file properties and manipulating files

java.io.File	
+File(pathname: String)	Creates a File object for the specified pathname. The pathname may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. child may be a filename or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of <i>hidden</i> is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period character '.'.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the pathname, resolves symbolic links (on Unix platforms), and converts drive letters to standard uppercase (on Win32 platforms).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\\book\\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\\book.
+lastModified(): long	Returns the time that the file was last modified.
+delete(): boolean	Deletes this file. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames this file. The method returns true if the operation succeeds.

Text I/O

- A **File** object encapsulates the properties of a file or a path, but *does not contain the methods for reading / writing data* from/to a file.
- In order to perform I/O, you need to create objects using appropriate Java I/O classes: **Scanner** and **PrintWriter**

Reading Data Using Scanner

java.util.Scanner

+Scanner(source: File)

Creates a Scanner that produces values scanned from the specified file.

+Scanner(source: String)

Creates a Scanner that produces values scanned from the specified string.

+close()

Closes this scanner.

+hasNext(): boolean

Returns true if this scanner has another token in its input.

+next(): String

Returns next token as a string.

+nextByte(): byte

Returns next token as a byte.

+nextShort(): short

Returns next token as a short.

+nextInt(): int

Returns next token as an int.

+nextLong(): long

Returns next token as a long.

+nextFloat(): float

Returns next token as a float.

+nextDouble(): double

Returns next token as a double.

+useDelimiter(pattern: String):
Scanner

Sets this scanner's delimiting pattern.


```
import java.util.Scanner;
public class ReadData {
    public static void main(String[] args) throws Exception{
        // Create a File instance
        java.io.File file = new java.io.File("scores.txt");
        // Create a Scanner for the file
        Scanner input = new Scanner(file);
        // Read data from a file
        while (input.hasNext()) {
            String firstName = input.next();
            int score = input.nextInt();
            System.out.println(Name + " " + score);
        }
        // Close the file
        input.close();
    }
}
```

Writing Data Using PrintWriter

java.io.PrintWriter

+PrintWriter(file: File)

+print(s: String): void

+print(c: char): void

+print(cArray: char[]): void

+print(i: int): void

+print(l: long): void

+print(f: float): void

+print(d: double): void

+print(b: boolean): void

Also contains the overloaded
println methods.

Also contains the overloaded
printf methods.

Creates a PrintWriter for the specified file.

Writes a string.

Writes a character.

Writes an array of character.

Writes an int value.

Writes a long value.

Writes a float value.

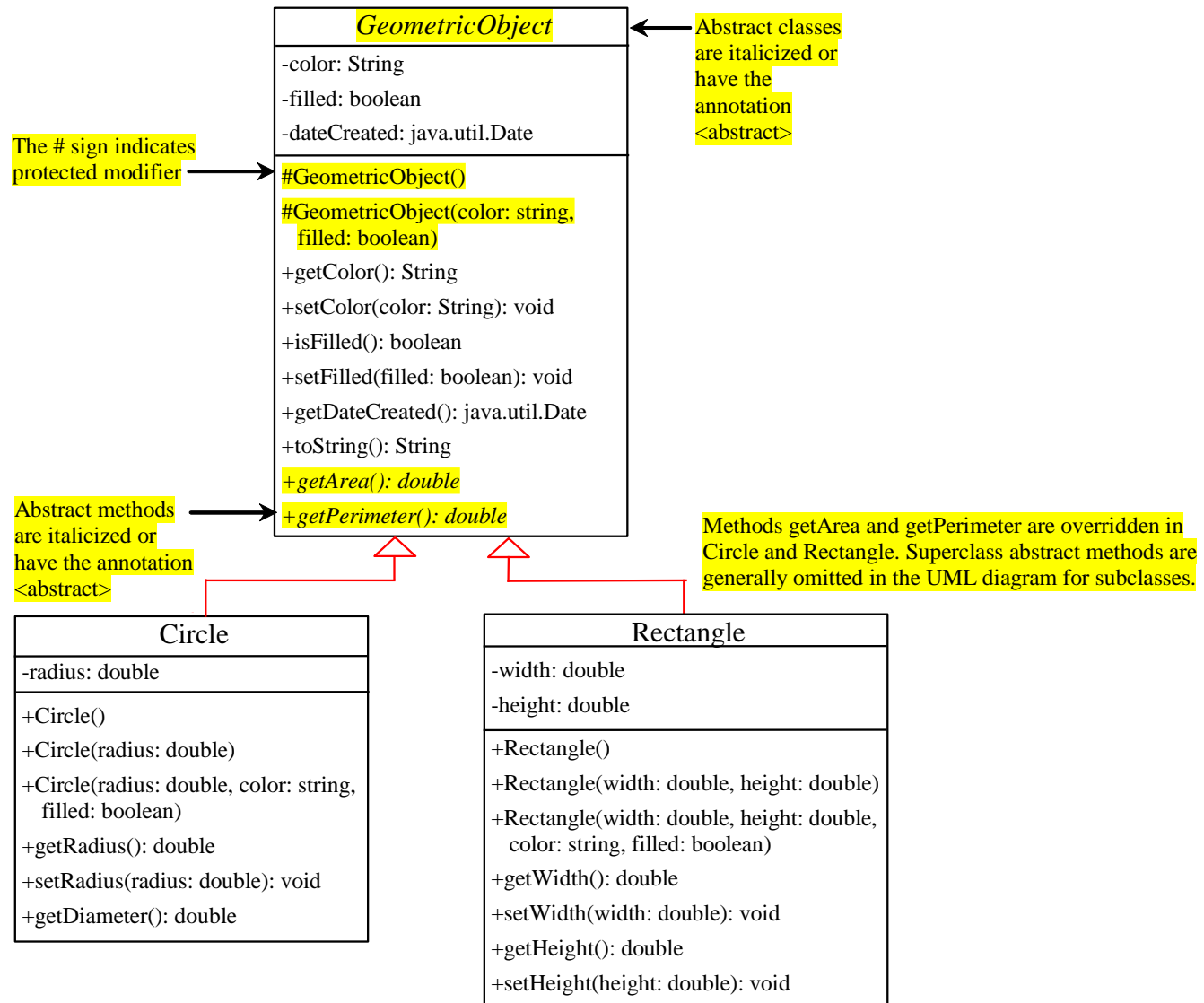
Writes a double value.

Writes a boolean value.

A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix. The printf method was introduced in §3.6, “Formatting Console Output and Strings.”

```
public class WriteData {  
    public static void main(String[] args)  
        throws Exception {  
        java.io.File file = new java.io.File("scores.txt");  
        if (file.exists()) {  
            System.out.println("File already exists");  
            System.exit(0);  
        }  
        // Create the file  
        java.io.PrintWriter output = new  
            java.io.PrintWriter(file);  
        // Write output to the file  
        output.print("John T Smith ");  
        // Close the file  
        output.close();  
    }  
}
```

Abstract Classes and Abstract Methods



```

public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;
    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }
    protected GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }
    public String getColor() {    return color;    }
    public void setColor(String color) {    this.color = color;    }
    public boolean isFilled() {    return filled;    }
    public void setFilled(boolean filled) {    this.filled = filled;    }
    public java.util.Date getDateCreated() {    return dateCreated;    }
    public String toString() {
        return "created on " + dateCreated + "\n color: " + color +
            " and filled: " + filled;
    }
    /** Abstract method getArea */
    public abstract double getArea();
    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}

```

```

public class Circle extends GeometricObject {
    private double radius;
    public Circle() { }
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
    public double getDiameter() {
        return 2 * radius;
    }
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    /* Print the circle info */
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() +
            " and the radius is " + radius);
    }
}

```

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle() { }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    public double getArea() {
        return width * height;
    }

    public double getPerimeter() {
        return 2 * (width + height);
    }
}
```

```

public class TestGeometricObject {
    public static void main(String[] args) {
        // Declare and initialize two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);
        System.out.println("The two objects have the same area? " +
            equalArea(geoObject1, geoObject2));
        // Display circle
        displayGeometricObject(geoObject1);
        // Display rectangle
        displayGeometricObject(geoObject2);
    }

    /** A method for comparing the areas of two geometric objects */
    public static boolean equalArea(GeometricObject object1,
        GeometricObject object2) {
        return object1.getArea() == object2.getArea();
    }

    /** A method for displaying a geometric object */
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println();
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }
}

```


abstract method in *abstract* class

- An abstract method cannot be contained in a nonabstract class.
- In a nonabstract subclass extended from an abstract class, all the abstract methods must be **implemented**, even if they are not used in the subclass.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract.

abstract classes

- An object cannot be created from abstract class
 - An abstract class cannot be instantiated using the new operator
 - We can still define its constructors, which are invoked in the constructors of its subclasses.
 - For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

abstract classes

- A class that contains abstract methods must be abstract
- An abstract class without abstract method:
 - It is possible to define an abstract class that contains no abstract methods.
 - We cannot create instances of the class using the new operator.
 - **This class is used as a base class for defining new subclasses**

abstract classes

- A subclass can be abstract even if its superclass is concrete.
- For example, the Object class is concrete, but a subclass, GeometricObject, is abstract

abstract classes

- A subclass can override a method from its superclass to define it abstract
 - rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass.
 - the subclass must be defined abstract

abstract classes as types

- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type:

```
GeometricObject c = new Circle(2);
```

- The following statement, which creates an **array** whose elements are of GeometricObject type, is correct

```
GeometricObject[] geo=new GeometricObject[10];
```

- There are only **null** elements in the array!!!

The *abstract* Calendar class and its GregorianCalendar subclass

- An instance of java.util.Date represents a specific instant in time with millisecond precision
- java.util.Calendar is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object for a specific calendar
 - Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.
 - java.util.GregorianCalendar is for the Gregorian calendar

The GregorianCalendar Class

- Java API for the GregorianCalendar class:
<http://docs.oracle.com/javase/8/docs/api/java/util/GregorianCalendar.html>
- new GregorianCalendar() constructs a default GregorianCalendar with the current time
- new GregorianCalendar(year, month, date) constructs a GregorianCalendar with the specified year, month, and date
 - The month parameter is 0-based, i.e., 0 is for January.

The *abstract* **Calendar** class and its **GregorianCalendar** subclass

java.util.Calendar

```
#Calendar()  
+get(field: int): int  
+set(field: int, value: int): void  
+set(year: int, month: int,  
    dayOfMonth: int): void  
+getActualMaximum(field: int): int  
+add(field: int, amount: int): void  
+getTime(): java.util.Date  
+setTime(date: java.util.Date): void
```

Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based, that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a Date object representing this calendar's time value (million second offset from the Unix epoch).

Sets this calendar's time with the given Date object.



java.util.GregorianCalendar

```
+GregorianCalendar()  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int)  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int,  
    hour: int, minute: int, second: int)
```

Constructs a GregorianCalendar for the current time.

Constructs a GregorianCalendar for the specified year, month, and day of month.

Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

The get Method in Calendar Class

- The get(int field) method defined in the Calendar class is useful to extract the date and time information from a Calendar object.
- The fields are defined as constants, as shown in the following.

Constant	Description
<u>YEAR</u>	The year of the calendar.
<u>MONTH</u>	The month of the calendar with 0 for January.
<u>DATE</u>	The day of the calendar.
<u>HOURL</u>	The hour of the calendar (12-hour notation).
<u>HOURL OF DAY</u>	The hour of the calendar (24-hour notation).
<u>MINUTE</u>	The minute of the calendar.
<u>SECOND</u>	The second of the calendar.
<u>DAY OF WEEK</u>	The day number within the week with 1 for Sunday.
<u>DAY OF MONTH</u>	Same as DATE.
<u>DAY OF YEAR</u>	The day number in the year with 1 for the first day of the year.
<u>WEEK OF MONTH</u>	The week number within the month.
<u>WEEK OF YEAR</u>	The week number within the year.
<u>AM PM</u>	Indicator for AM or PM (0 for AM and 1 for PM).

```

import java.util.*;
public class TestCalendar {
    public static void main(String[] args) {
        // Construct a Gregorian calendar for the current date and time
        Calendar calendar = new GregorianCalendar();
        System.out.println("Current time is " + new Date());
        System.out.println("YEAR:\t" + calendar.get(Calendar.YEAR));
        System.out.println("MONTH:\t" + calendar.get(Calendar.MONTH));
        System.out.println("DATE:\t" + calendar.get(Calendar.DATE));
        System.out.println("HOUR:\t" + calendar.get(Calendar.HOUR));
        System.out.println("HOUR_OF_DAY:\t" + calendar.get(Calendar.HOUR_OF_DAY));
        System.out.println("MINUTE:\t" + calendar.get(Calendar.MINUTE));
        System.out.println("SECOND:\t" + calendar.get(Calendar.SECOND));
        System.out.println("DAY_OF_WEEK:\t" + calendar.get(Calendar.DAY_OF_WEEK));
        System.out.println("DAY_OF_MONTH:\t" + calendar.get(Calendar.DAY_OF_MONTH));
        System.out.println("DAY_OF_YEAR: " + calendar.get(Calendar.DAY_OF_YEAR));
        System.out.println("WEEK_OF_MONTH: " + calendar.get(Calendar.WEEK_OF_MONTH));
        System.out.println("WEEK_OF_YEAR: " + calendar.get(Calendar.WEEK_OF_YEAR));
        System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
        // Construct a calendar for January 1, 2020
        Calendar calendar1 = new GregorianCalendar(2020, 0, 1);
        System.out.println("January 1, 2020 is a " +
            dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)) );
    }
    public static String dayNameOfWeek(int dayOfWeek) {
        switch (dayOfWeek) {
            case 1: return "Sunday"; case 2: return "Monday"; case 3: return "Tuesday";
            ... case 7: return "Saturday";
            default: return null;
        }
    }
}

```

Interfaces

- What is an interface?
 - An interface is a class-like construct that contains only constants and abstract methods.
- Why is an interface useful?
 - An interface is similar to an abstract class, but the intent of an interface is to **specify behavior** for objects.
 - For example: specify that the objects are **comparable, edible, cloneable, ...**
 - Allows multiple inheritance.

Define an Interface

- Declaration:

```
public interface InterfaceName {  
    // constant declarations;  
    // method signatures;  
}
```

- Example:

```
public interface Edible {  
    /** Describe how to eat */  
    public abstract String howToEat();  
}
```

Interface is a Special Class

- An interface is treated like a special class in Java:
 - Each interface is compiled into a separate bytecode file, just like a regular class.
 - Like an abstract class, you cannot create an instance from an interface using the new operator
 - Uses:
 - as a data type for a variable
 - as the result of casting

Interface Example

- The Edible interface specifies whether an object is edible

```
public interface Edible {  
    public abstract String howToEat();  
}
```

- The class Chicken implements the Edible interface:

```
class Chicken extends Animal implements Edible {  
    public String howToEat() {  
        return "Chicken: Fry it";  
    }  
}
```

```

interface Edible {
    public abstract String howToEat(); /** Describe how to eat */
}
class Animal { }
class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: Fry it";
    }
}
class Tiger extends Animal { /** Does not implement Edible */
}
abstract class Fruit implements Edible { }
class Apple extends Fruit {
    public String howToEat() {
        return "Apple: Make apple cider";
    }
}
class Orange extends Fruit {
    public String howToEat() {
        return "Orange: Make orange juice";
    }
}
public class TestEdible {
    public static void main(String[] args) {
        Object[] objects = {new Tiger(), new Chicken(), new Apple()};
        for (int i = 0; i < objects.length; i++)
            if (objects[i] instanceof Edible)
                System.out.println(((Edible)objects[i]).howToEat());
    }
}

```


Omitting Modifiers in Interfaces

- All data fields are *public static final* in an interface
- All methods are *public abstract* in an interface
- These modifiers can be omitted:

```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

Equivalent

```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

- A constant defined in an interface can be accessed using InterfaceName.CONSTANT_NAME, for example: T1.K

Example: The Comparable Interface

```
// This interface is defined in  
// the java.lang package
```

```
package java.lang;  
public interface Comparable {  
    int compareTo (Object o) ;  
}
```

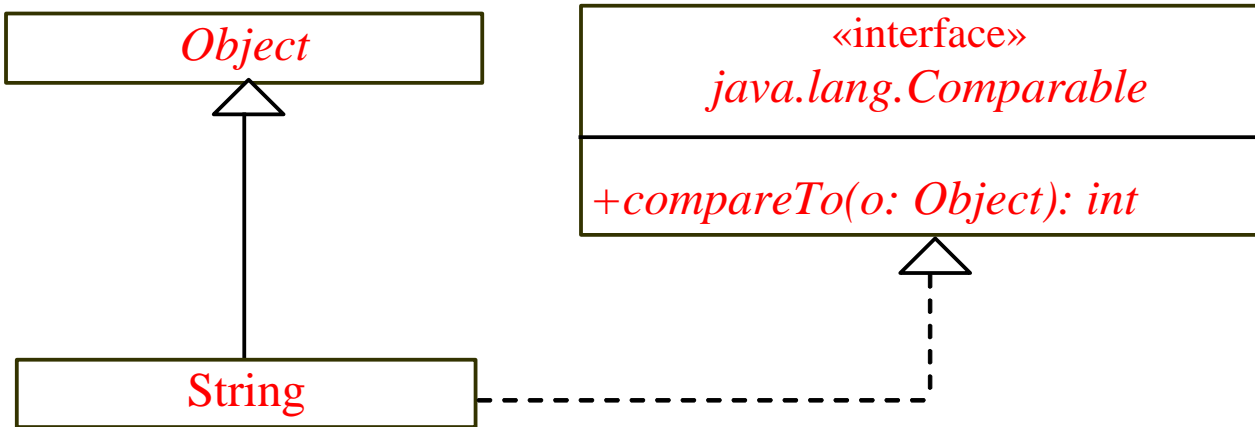
String and Date Classes

- Many classes (e.g., String and Date) in the Java library implement Comparable to define **a natural order** for the objects

```
public class String extends Object  
    implements Comparable {  
    // class body omitted  
}
```

```
public class Date extends Object  
    implements Comparable {  
    // class body omitted  
}
```

<code>new String()</code>	<code>instanceof String</code>	true
<code>new String()</code>	<code>instanceof Comparable</code>	true
<code>new java.util.Date()</code>	<code>instanceof java.util.Date</code>	true
<code>new java.util.Date()</code>	<code>instanceof Comparable</code>	true



In UML, the interface and the methods are italicized

dashed lines and triangles are used to point to the interface

Generic max Method

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum of two objects */
    public static Comparable max
        (Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
        }
    }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum of two objects */
    public static Object max
        (Object o1, Object o2) {
        if (((Comparable)o1).compareTo(o2) > 0)
            return o1;
        else
            return o2;
        }
    }
}
```

(b)

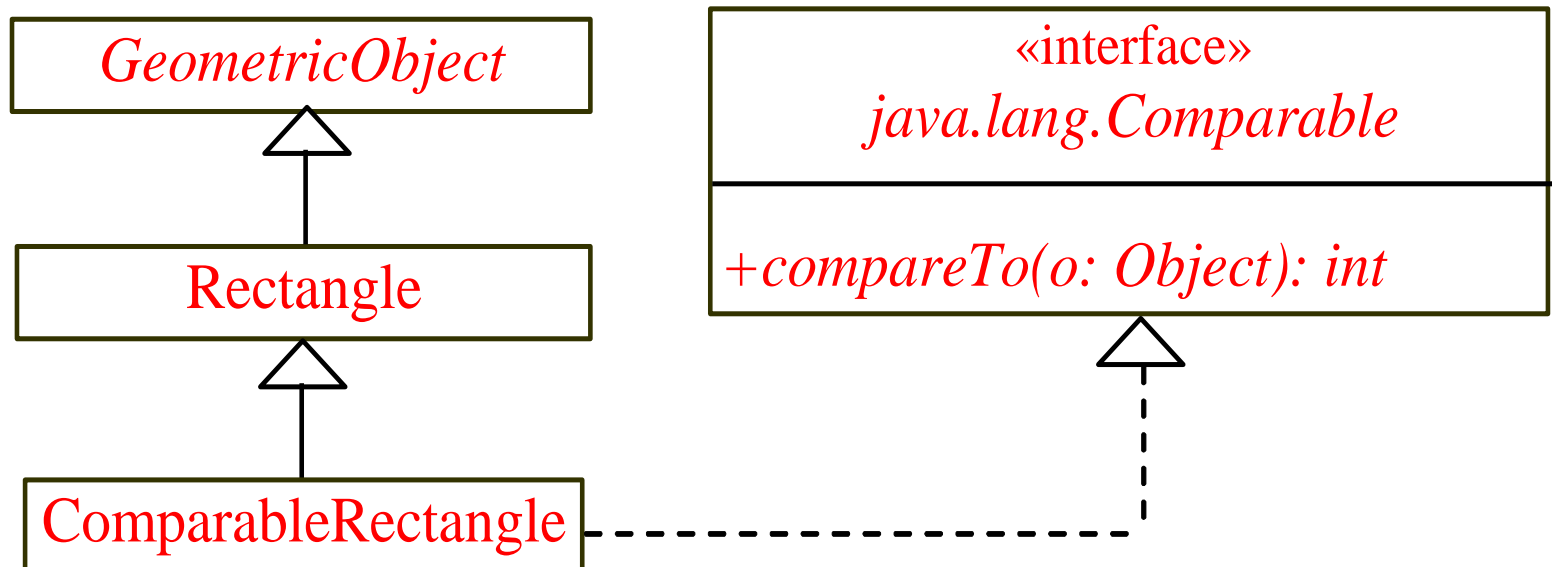
```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, we need to cast it to String or Date explicitly.

Defining Classes to Implement Comparable

- We cannot use the max method to find the larger of two instances of Rectangle, because Rectangle does not implement Comparable
- We can define a new rectangle class ComparableRectangle that implements Comparable: the instances of this new class are comparable



```

public class ComparableRectangle extends Rectangle
    implements Comparable {
    /** Construct a ComparableRectangle with specified properties */
    public ComparableRectangle(double width, double height) {
        super(width, height);
    }

    /** Implement the compareTo method defined in Comparable */
    public int compareTo(Object o) {
        if (getArea() > ((ComparableRectangle)o).getArea())
            return 1;
        else if (getArea() < ((ComparableRectangle)o).getArea())
            return -1;
        else
            return 0;
    }

    public static void main(String[] args) {
        ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
        ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
        System.out.println(Max.max(rectangle1, rectangle2));
    }
}

```

The Cloneable Interface

- Marker Interface: an empty interface
 - Does not contain constants or methods
 - It is used to denote that a class possesses certain desirable properties
- A class that implements the Cloneable interface is marked cloneable: its objects can be cloned using the clone() method defined in the Object class

```
package java.lang;  
public interface Cloneable {  
}
```


The Cloneable Interface

- Calendar (in the Java library) implements Cloneable:

```
Calendar calendar = new GregorianCalendar(2020, 1, 1);  
Calendar calendarCopy = (Calendar)(calendar.clone());  
System.out.println("calendar == calendarCopy is "  
    + (calendar == calendarCopy));
```

Displays:

calendar == calendarCopy is false

```
System.out.println("calendar.equals(calendarCopy) is "  
    + calendar.equals(calendarCopy));
```

Displays:

calendar.equals(calendarCopy) is true

Implementing the **Cloneable** Interface

- If we try to create a clone of an object instance of a class that does not implement the **Cloneable** interface, it throws **CloneNotSupportedException**
- We can override the **clone()** method from the **Object** class to create custom clones
 - The **clone** method in the **Object** class creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned.
- The **clone()** method returns an **Object** that needs to be casted

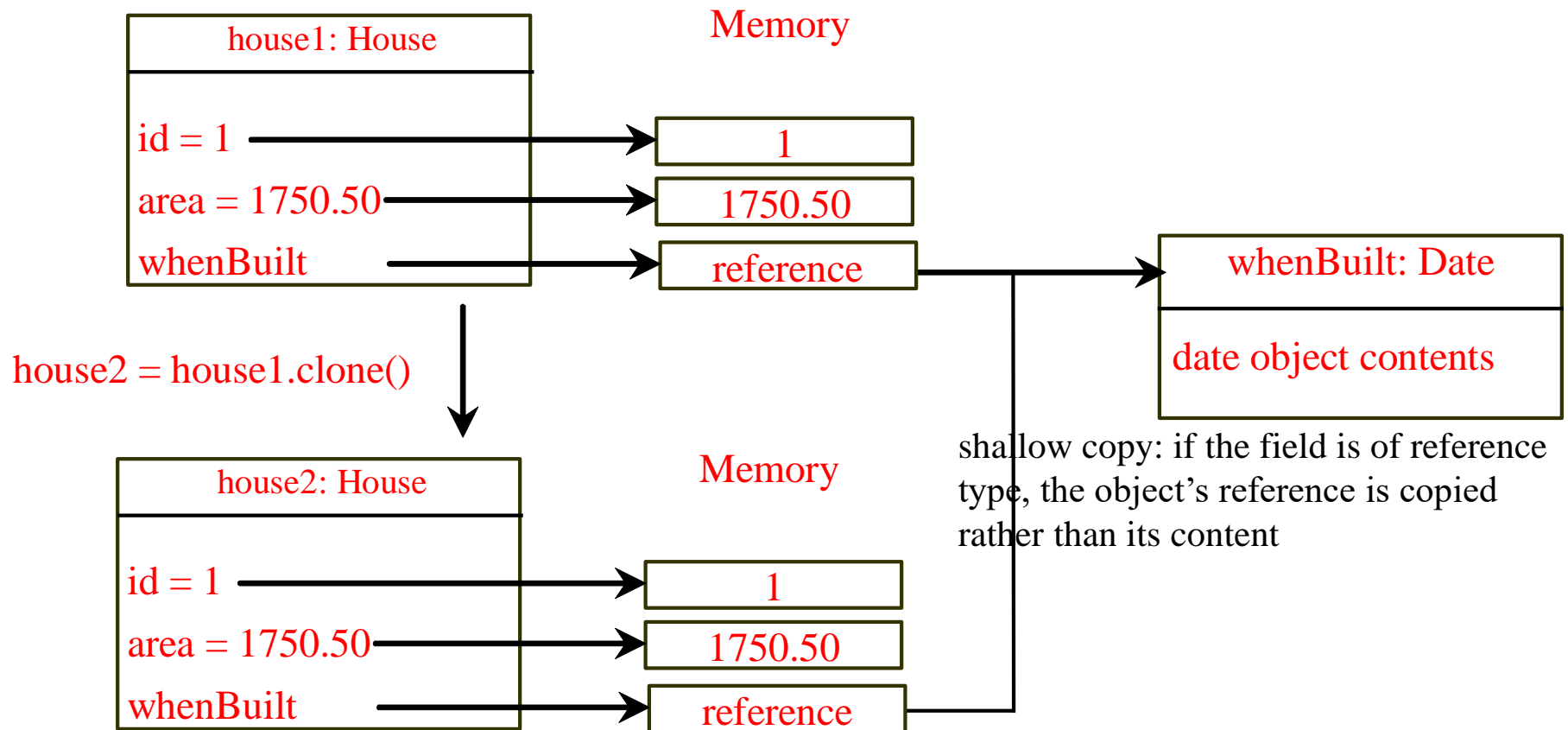
```

public class House implements Cloneable, Comparable {
    private int id;
    private double area;
    private java.util.Date whenBuilt;
    public House(int id, double area) {this.id = id; this.area = area;
        whenBuilt = new java.util.Date();}
    public double getId() { return id;}
    public double getArea() { return area;}
    public java.util.Date getWhenBuilt() { return whenBuilt;}
    /** Override the protected clone method defined in the Object
        class, and strengthen its accessibility */
    public Object clone() {
        try {
            return super.clone();
        }catch (CloneNotSupportedException ex) {
            return null;
        }
    }
    /** Implement the compareTo method defined in Comparable */
    public int compareTo(Object o) {
        if (area > ((House)o).area)
            return 1;
        else if (area < ((House)o).area)
            return -1;
        else
            return 0;
    }
}

```

Shallow vs. Deep Copy

- `House house1 = new House(1, 1750.50);`
- `House house2 = (House)(house1.clone());`



For deep copying, we can override the clone method with custom object creation

```
public class House implements Cloneable {  
    ...  
    public Object clone() { // deep copy  
        try {  
            House h = (House) (super.clone());  
            h.whenBuilt = (Date) (whenBuilt.clone());  
            return h;  
        } catch (CloneNotSupportedException ex) {  
            return null;  
        }  
    }  
    ...  
}
```

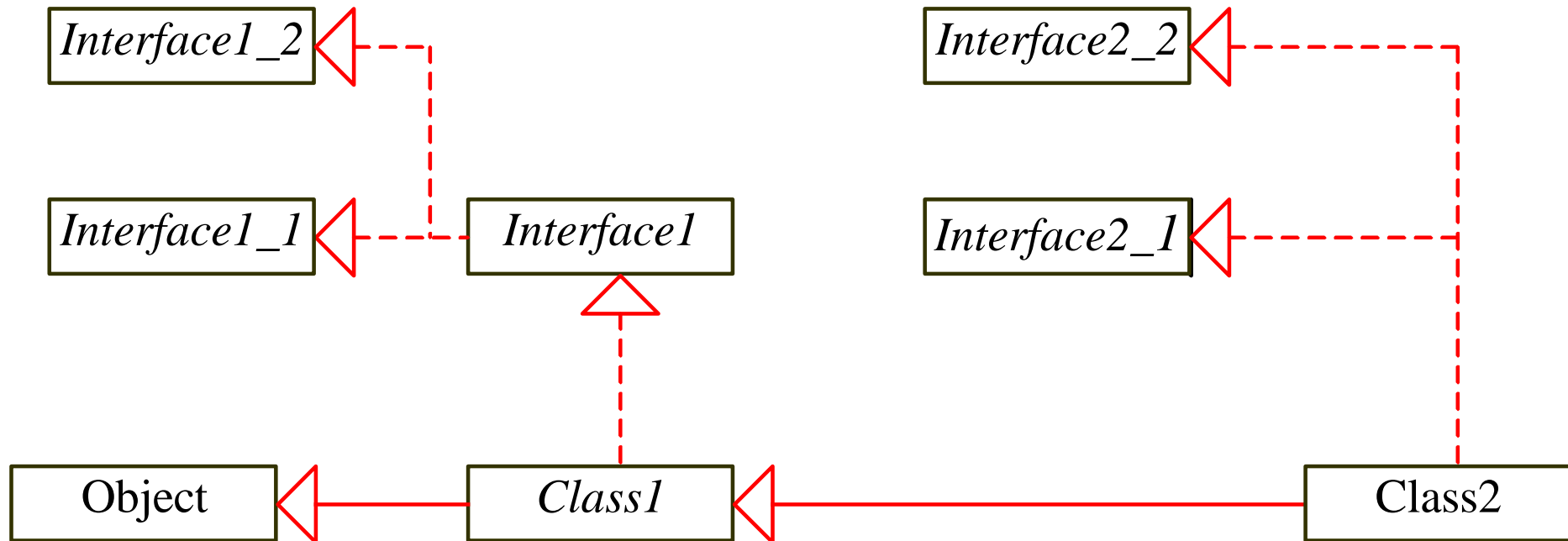
Interfaces vs. Abstract Classes

- In an interface, the data must be constants; an abstract class can have all types of data
- Each method in an interface has only a signature without implementation; an abstract class can have concrete methods

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining . An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be <u>public</u> <u>static</u> <u>final</u>	No constructors. An interface cannot be instantiated using the new operator.	All methods must be <u>public</u> <u>abstract</u> methods

Interfaces vs. Abstract Classes

- A class can implement any number of interfaces
- An interface can extend another interface
- There is no root for interfaces



Interfaces/Abstract classes & Polymorphism

- Which of these (Interfacesⁱ, Abstract^a and Concrete^c classes):
 - can have instance variables? ^{ac}
 - can have static variables? ^{ac}
 - can have static final constants? ^{iac}
 - can have constructors? ^{ac}
 - can have abstract methods? ^{ia}
 - can have concrete methods? ^{ac}
 - can be constructed? ^c
- These are common interview questions.

Caution: conflicting interfaces

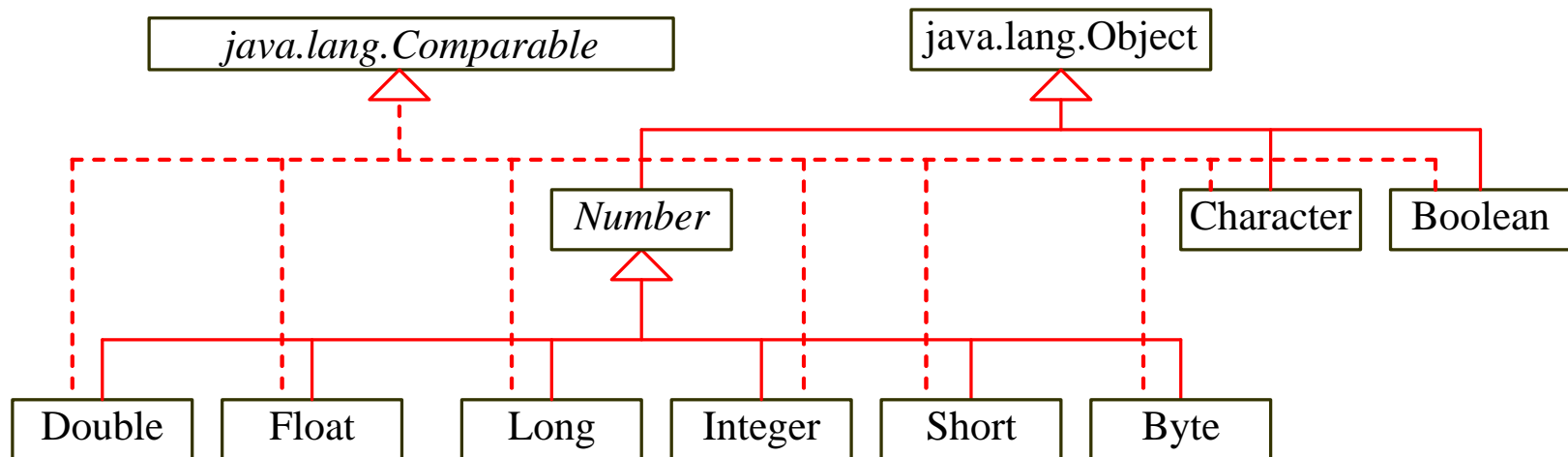
- Errors detected by the compiler:
 - If a class implements two interfaces with conflict information:
 - two same constants with different values,
or
 - two methods with same signature but
different return type

Whether to use an interface or a class?

- Strong is-a: a relationship that clearly describes a parent-child relationship - **should be modeled using classes and class inheritance**
 - For example: a staff member is a person
- Weak is-a (is-kind-of): indicates that an object possesses a certain property - **should be modeled using interfaces**
 - For example: all strings are comparable, so the String class implements the Comparable interface
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired

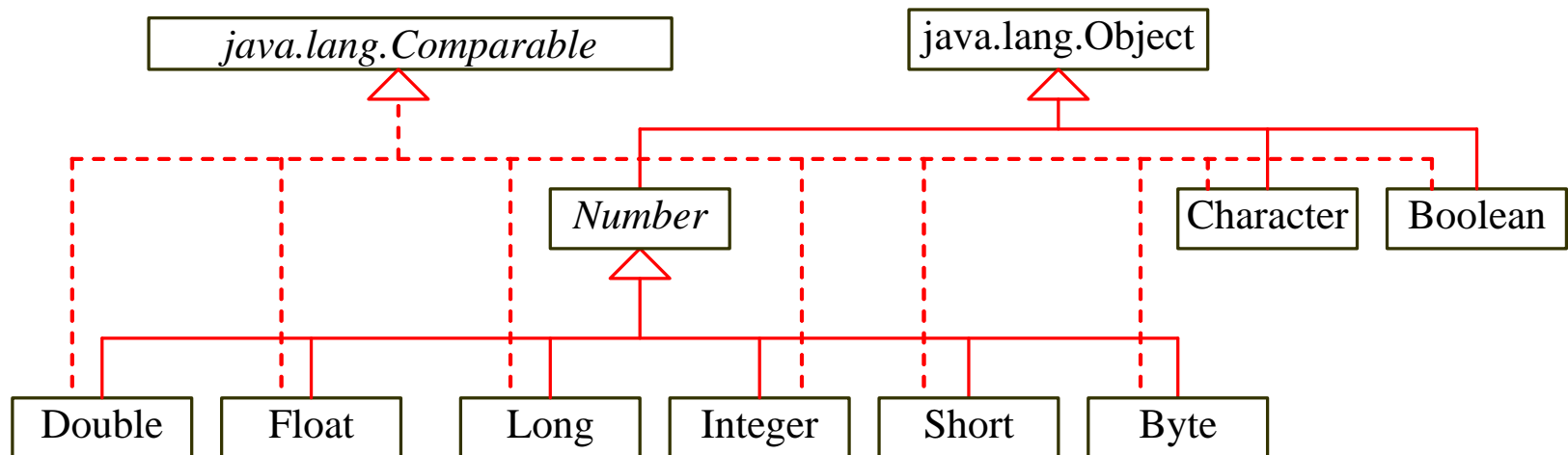
Wrapper Classes

- Primitive data types in Java → **Better performance**
- Each primitive has a wrapper class: Boolean, Character, Short, Byte, Integer, Long, Float, Double
 - The wrapper classes do not have no-arg constructors
 - The instances of all wrapper classes are immutable: their internal values cannot be changed once the objects are created



Wrapper Classes

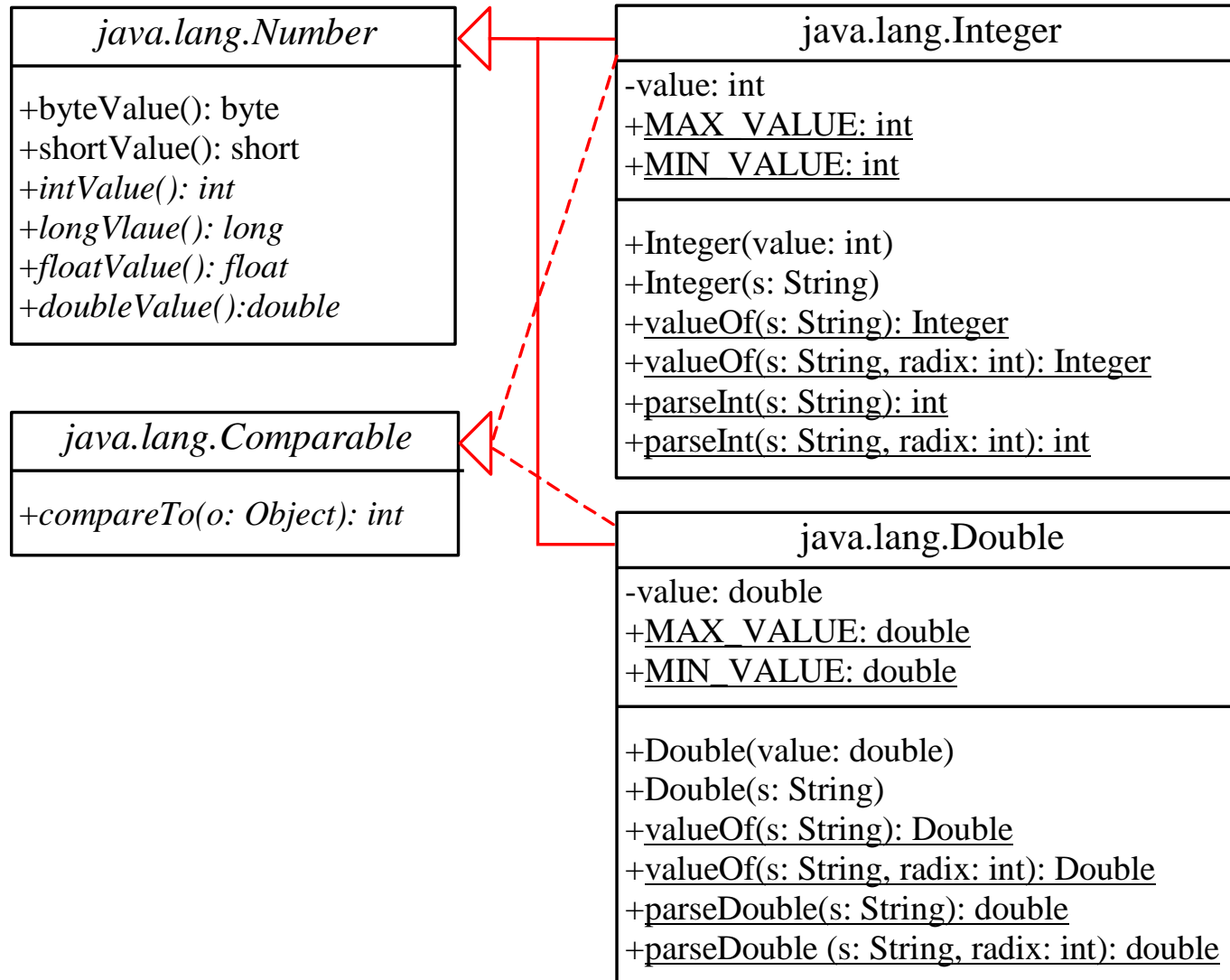
- Each wrapper class overrides the `toString`, `equals`, and `hashCode` methods defined in the `Object` class
- Since these classes implement the `Comparable` interface, the `compareTo` method is implemented in these classes



The Number Class

- Each numeric wrapper class extends the abstract Number class:
 - The abstract Number class contains the methods doubleValue, floatValue, intValue, longValue, shortValue, and byteValue to “convert” objects into primitive type values
 - The methods doubleValue, floatValue, intValue, longValue are abstract
 - The methods byteValue and shortValue are not abstract, which simply return (byte)intValue() and (short)intValue(), respectively
- Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue and longValue

The Integer and Double Classes



Wrapper Classes

- You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value
- The constructors for Integer and Double are:

```
public Integer(int value)
```

```
public Integer(String s)
```

```
public Double(double value)
```

```
public Double(String s)
```

Numeric Wrapper Class Constants

- Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE:
 - MAX_VALUE represents the maximum value of the corresponding primitive data type
 - For Float and Double, MIN_VALUE represents the minimum *positive* float and double values
 - The maximum integer: 2,147,483,647
 - The minimum positive float: 1.4E-45
 - The maximum double floating-point number: 1.79769313486231570e+308d

The Static valueOf Methods

- The numeric wrapper classes have a static method `valueOf(String s)` to create a new object initialized to the value represented by the specified string:

```
Double doubleObject = Double.valueOf("12.4");
```

```
Integer integerObject = Integer.valueOf("12");
```

- Each numeric wrapper class has overloaded parsing methods to parse a numeric string into an appropriate numeric value:

```
double d = Double.parseDouble("12.4");
```

```
int i = Integer.parseInt("12");
```

Sorting an Array of Objects

```
public class GenericSort {
    public static void main(String[] args) {
        Integer[] intArray={new Integer(2),new Integer(4),new Integer(3)};
        sort(intArray);
        printList(intArray);
    }
    public static void sort(Object[] list) {
        Object currentMax;
        int currentMaxIndex;
        for (int i = list.length - 1; i >= 1; i--) {
            currentMax = list[i];
            currentMaxIndex = i; // Find the maximum in the list[0..i]
            for (int j = i - 1; j >= 0; j--) {
                if (((Comparable)currentMax).compareTo(list[j]) < 0) {
                    currentMax = list[j];
                    currentMaxIndex = j;
                }
            }
            list[currentMaxIndex] = list[i];
            list[i] = currentMax;
        }
    }
    public static void printList(Object[] list) {
        for (int i=0;i<list.length;i++) System.out.print(list[i]+" ");}}
```

The objects are instances of the Comparable interface and they are compared using the compareTo method.

Sorting an Array of Objects

- Java provides a static sort method for sorting an array of Object in the java.util.Arrays class:

```
java.util.Arrays.sort(intArray) ;
```

Arrays of Objects

- Arrays are objects:
 - An array is an instance of the Object class
 - If A is a subclass of B, every instance of A[] is an instance of B[]
- ```
new int[10] instanceof Object true
new GregorianCalendar[10] instanceof Calendar[] ; true
new Calendar[10] instanceof Object[] true
new Calendar[10] instanceof Object true
```
- Although an int value can be assigned to a double type variable, int[] and double[] are two incompatible types:
    - We cannot assign an int[] array to a variable of double[] array

# Wrapper Classes

- Automatic Conversion Between Primitive Types and Wrapper Class Types:
  - JDK 1.5 allows primitive type and wrapper classes to be converted automatically = boxing

```
Integer[] intArray = {new Integer(2),
new Integer(4), new Integer(3)};
```

(a)

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(b)

New JDK 1.5 boxing

```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

# BigInteger and BigDecimal

- BigInteger and BigDecimal classes in the java.math package:
  - For computing with very large integers or high precision floating-point values
    - BigInteger can represent an integer of any size
    - BigDecimal has no limit for the precision (as long as it's finite=terminates)
  - Both are *immutable*
  - Both extend the Number class and implement the Comparable interface.

# BigInteger and BigDecimal

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

**18446744073709551614**

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

**0.333333333333333333333334**

# BigInteger and BigDecimal

```
import java.math.*;

public class LargeFactorial {
 public static void main(String[] args) {
 System.out.println("50! is \n" + factorial(50));
 }
 public static BigInteger factorial(long n) {
 BigInteger result = BigInteger.ONE;
 for (int i = 1; i <= n; i++)
 result = result.multiply(new BigInteger(i+""));
 return result;
 }
}
```

30414093201713378043612608166064768844377641  
5689605120000000000000