Unit II: INHERITANCE, POLYMORPHISM AND EXCEPTIONS

Dr. S. Kavi Priya, ASP/CSE, MSEC
urskavi@mepcoeng.ac.in
9842295563

Review: Classes

- User-defined data types
  - Defined using the “class” keyword
  - Each class has associated
    - Data members (any object type)
    - Methods that operate on the data
- New instances of the class are declared using the “new” keyword
- “Static” members/methods have only one copy, regardless of how many instances are created

Example: Shared Functionality

```
public class Student {
   String name;
   char gender;
   Date birthday;
   Vector<Grade> grades;

   double getGPA() {
      …
   }

   int getAge(Date today) {
      …
   }
}
```

```
public class Professor {
   String name;
   char gender;
   Date birthday;
   Vector<Paper> papers;

   int getCiteCount() {
      …
   }

   int getAge(Date today) {
      …
   }
}
```

```java
public class Person {
   String name;
   char gender;
   Date birthday;

   int getAge(Date today) {
      …
   }
}
```

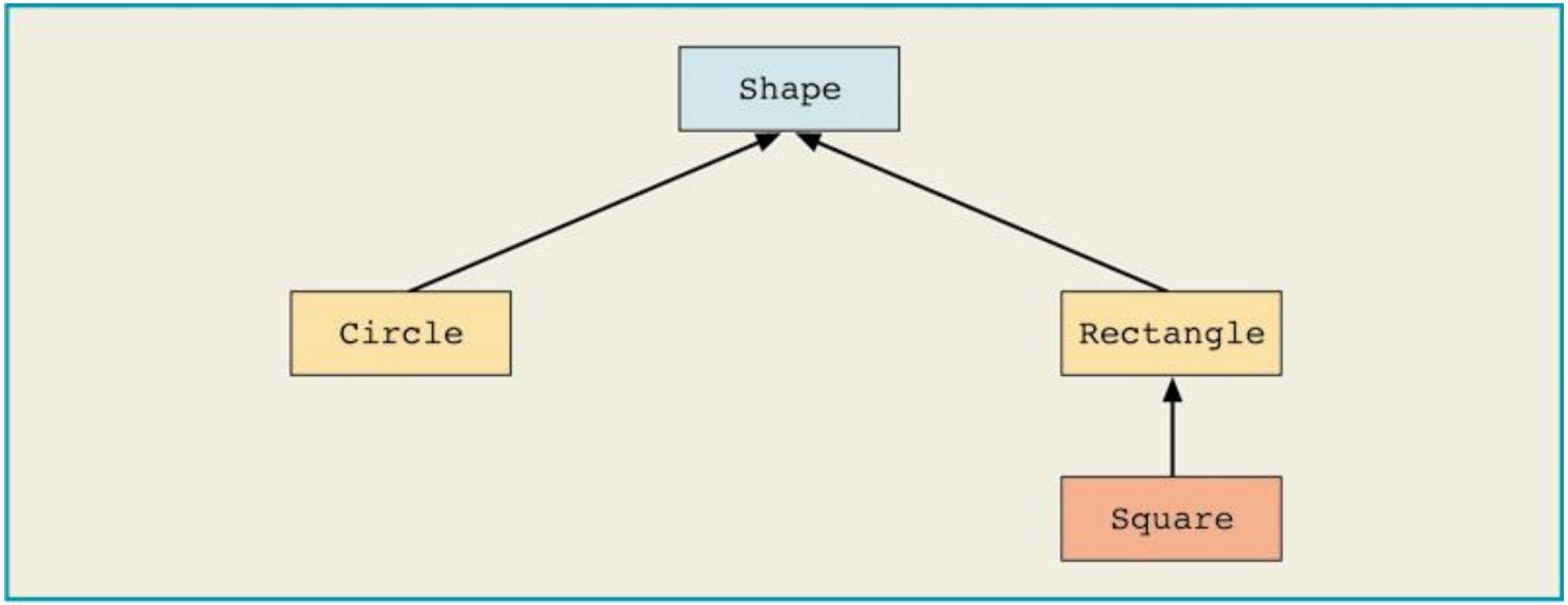
```java
public class Student
      extends Person {

   Vector<Grade> grades;

   double getGPA() {
      …
   }
}
```

```java
public class Professor
      extends Person {

   Vector<Paper> papers;

   int getCiteCount() {
      …
   }
}
```

Inheritance

- "is-a" relationship
- Single inheritance:
  - Subclass is derived from one existing class (superclass)
- Multiple inheritance:
  - Subclass is derived from more than one superclass
  - Not supported by Java
  - A class can only extend the definition of one class

Inheritance (Contd.)



🟩 Inheritance hierarchy

```
modifier(s) class ClassName extends ExistingClassName
                                          modifier(s)
{
    memberList
}
```

Inheritance

`class Circle` Derived from `class Shape`

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

Inheritance

- Allow us to specify *relationships between types*
  - Abstraction, generalization, specification
  - The “is-a” relationship
  - Examples?

- Why is this useful in programming?
  - Allows for code reuse
  - More intuitive/expressive code

Code Reuse

- General functionality can be written once and applied to **any* subclass
- Subclasses can specialize by adding members and methods, or overriding functions

Inheritance: Adding Functionality

- Subclasses have ***all*** of the data members and methods of the superclass
- Subclasses can add to the superclass
  - Additional data members
  - Additional methods
- Subclasses are more specific and have more functionality
- Superclasses capture generic functionality common across many types of objects

```java
public class Person {
   String name;
   char gender;
   Date birthday;

   int getAge(Date today) {
      …
   }
}
```

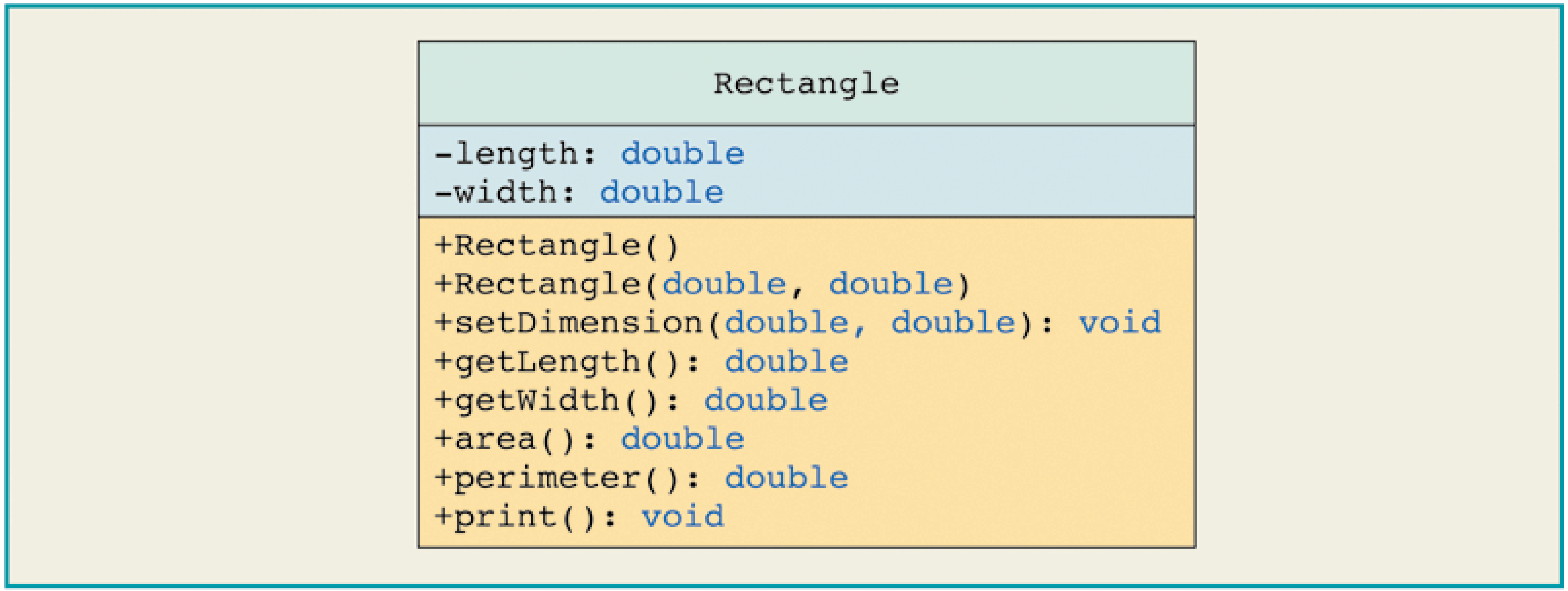
```java
public class Student
      extends Person {

   Vector<Grade> grades;

   double getGPA() {
      …
   }
}
```

```java
public class Professor
      extends Person {

   Vector<Paper> papers;

   int getCiteCount() {
      …
   }
}
```

Brainstorming

- What are some other examples of possible inheritance hierarchies?
  - Person -> student, faculty…
  - Shape -> circle, triangle, rectangle…
  - Other examples???

UML Diagram: `Rectangle`



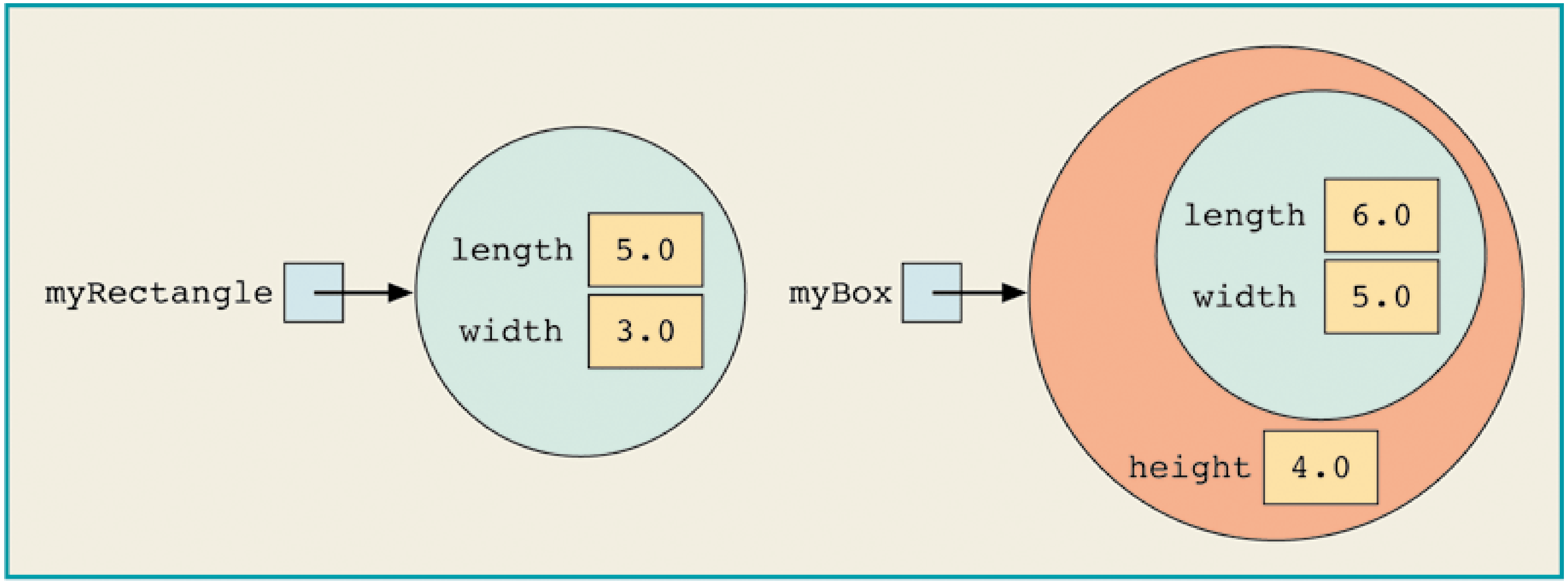
```
Rectangle
-length: double
-width: double
+Rectangle()
+Rectangle(double, double)
+setDimension(double, double): void
+getLength(): double
+getWidth(): double
+area(): double
+perimeter(): double
+print(): void
```

🟩 UML class diagram of the **class** Rectangle

What if we want to implement a 3d box object?

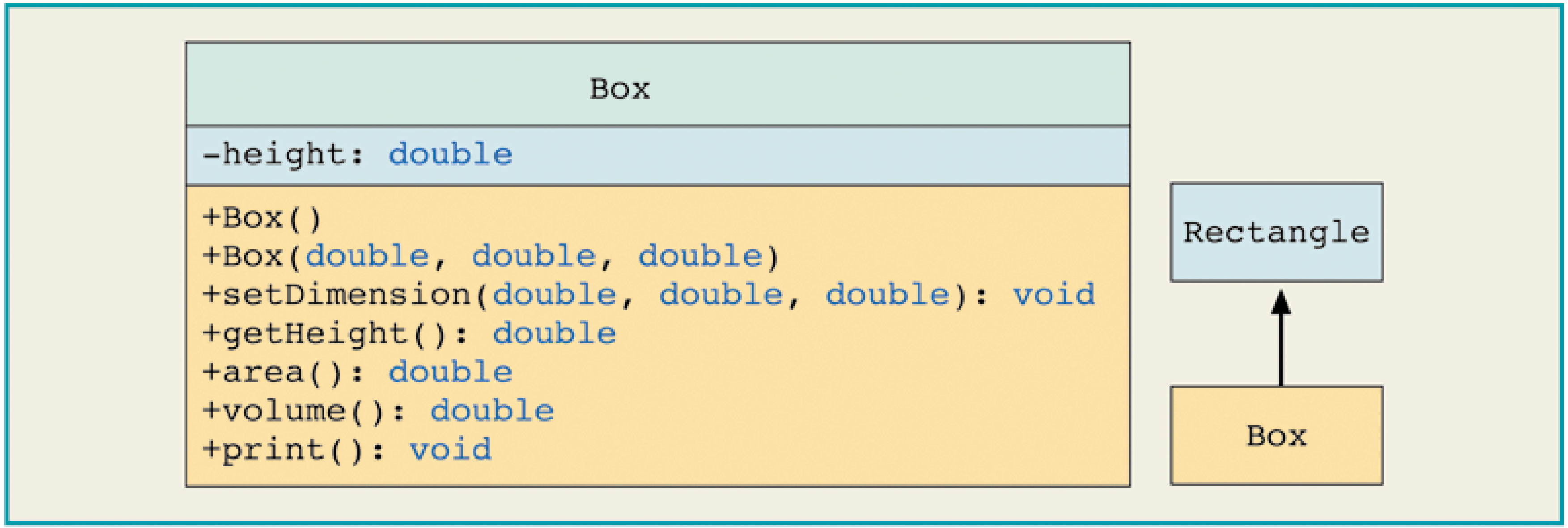
Objects `myRectangle` and `myBox`

```
Rectangle myRectangle = new Rectangle(5, 3);
Box myBox = new Box(6, 5, 4);
```



🟩 Objects myRectangle and myBox

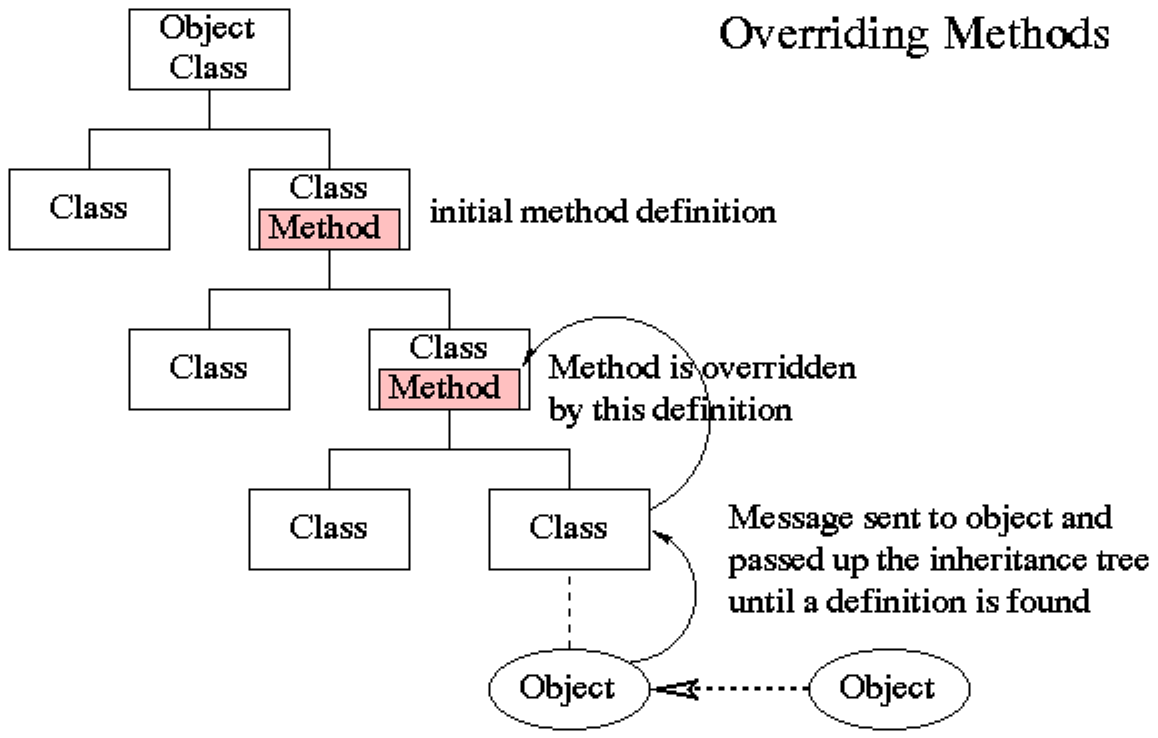
UML Class Diagram: class Box



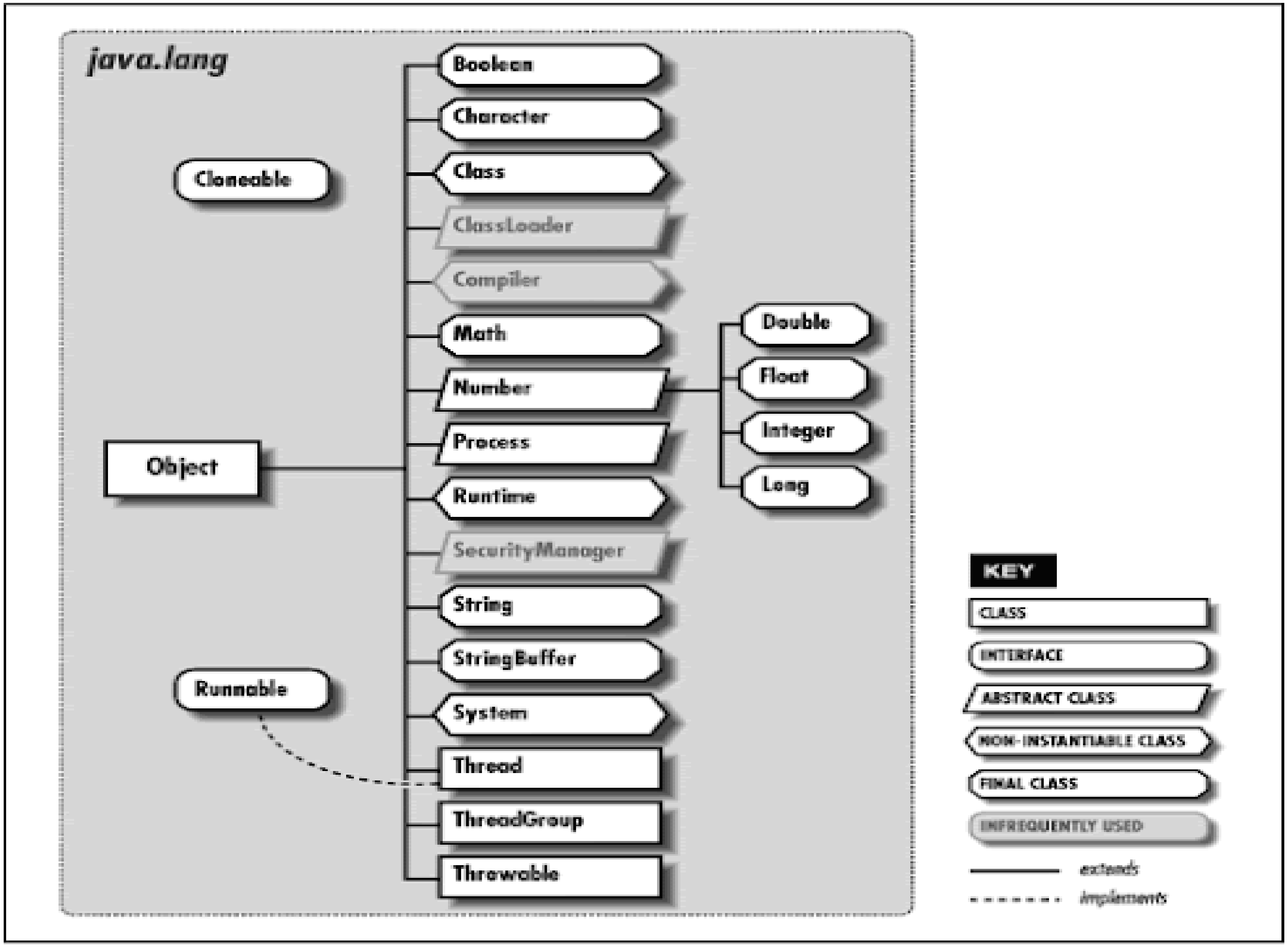
UML class diagram of the class Box and the inheritance hierarchy

Both a Rectangle and a Box have a surface area, but they are computed differently

Class Hierarchy


Object Class
Overriding Methods
Class
Class
Method
initial method definition
Class
Class
Method
Method is overridden by this definition
Class
Class
Message sent to object and passed up the inheritance tree until a definition is found
Object
Object

Example Hierarchy



Overriding Methods

- A subclass can override (redefine) the methods of the superclass
  - Objects of the subclass type will use the new method
  - Objects of the superclass type will use the original

class Rectangle

```
public double area()
{
      return  getLength() * getWidth();
}
```

class Box

```
public double area()
{
      return  2 * (getLength() * getWidth()
                + getLength() * height
                + getWidth() * height);
}
```

final Methods

- Can declare a method of a class final using the keyword `final`

```
public final void doSomeThing()
{
        //...
}
```

- If a method of a `class` is declared `final`, it cannot be overridden with a new definition in a derived class

Calling methods of the superclass

- To write a method's definition of a subclass, specify a call to the public method of the superclass
  - If subclass overrides public method of superclass, specify call to public method of superclass:

    ```
    super.MethodName(parameter list)
    ```

  - If subclass does not override public method of superclass, specify call to public method of superclass:

    ```
    MethodName(parameter list)
    ```

class Box

```
public void setDimension(double l, double w, double h)
{
    super.setDimension(l, w);
    if (h >= 0)
        height = h;
    else
        height = 0;
}}
```

Box overloads the method setDimension
(Different parameters)

Defining Constructors of the Subclass

- Call to constructor of superclass:
  - Must be first statement
  - Specified by super parameter list

```
public Box()
{
    super();
    height = 0;
}

public Box(double l, double w, double h)
{
    super(l, w);
    height = h;
}
```

Access Control

- Access control keywords define which classes can access classes, methods, and members

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
none	Y	Y	N	N
private	Y	N	N	N

instanceof

- The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The instanceof in java is also known as type comparison operator because it compares the instance with type.
- It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.
- **Example1:**

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true   }  }
```

```
Output: true
```

instanceof

Example2:

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal
   public static void main(String args[]){
 Dog1 d=new Dog1();
Animal a1=new Animal();
Animal a2=new Dog1();
 System.out.println(d instanceof Animal);//true
System.out.println(a1 instanceof Dog1);//false
System.out.println(a2 instanceof Dog1);//true
 System.out.println(a2 instanceof Animal);//true
System.out.println(a2 instanceof Object);//true
} }
```

```
Output: true    false         true       true       true
```

instanceof

Example3:

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
System.out.println(d instanceof Object);//true
  System.out.println(d instanceof Dog2);//false
} }
```

```
Output:   true
          null
```

ok downcasting performed

Downcasting with java instanceof operator

- `Dog d=new Animal();//Compilation error`

```
Dog d=(Dog)new Animal(); //Compiles but ClassCastException thrown at runtime
```

DOWNCASTING

```
class Animal { }
class Dog3 extends Animal {
 static void method(Animal a) {
   if(a instanceof Dog3){
      Dog3 d=(Dog3)a;//downcasting
      System.out.println("ok downcasting performed");     }   }
    public static void main (String [] args) {
   Animal a=new Dog3();
   Dog3.method(a);    }  }
```

OUTPUT: ok downcasting performed

Downcasting without the use of java instanceof

```
class Animal { }
class Dog4 extends Animal {
  static void method(Animal a) {
       Dog4 d=(Dog4)a;//downcasting
       System.out.println("ok downcasting performed");
  }
   public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
  }
}
```

Output:ok downcasting performed

Polymorphism

- Can treat an object of a subclass as an object of its superclass
  - A reference variable of a superclass type can point to an object of its subclass

```
Person name, nameRef;
PartTimeEmployee employee, employeeRef;
name = new Person("John", "Blair");
employee = new PartTimeEmployee("Susan", "Johnson",
                                12.50, 45);
nameRef = employee;
System.out.println("nameRef: " + nameRef);
```

```
nameRef: Susan Johnson wages are: $562.5
```

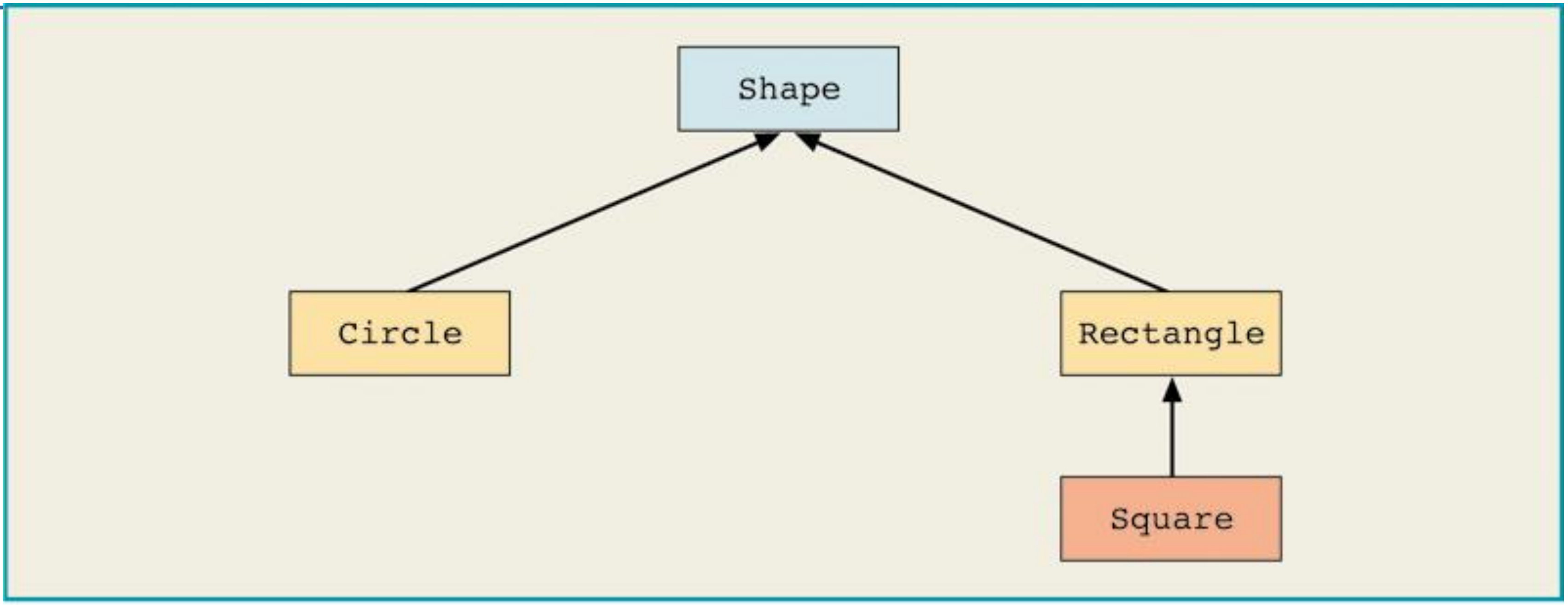
Polymorphism

- Late binding or dynamic binding (run-time binding):
  - Method to be executed is determined at execution time, not compile time
- Polymorphism: to assign multiple meanings to the same method name
- Implemented using late binding

Polymorphism (continued)

- The reference variable `name` or `nameRef` can point to any object of the `class Person` or the `class PartTimeEmployee`
- These reference variables have many forms, that is, they are polymorphic reference variables
- They can refer to objects of their own class or to objects of the classes inherited from their class

Polymorphism and References



🟩 Inheritance hierarchy

```
Shape myShape = new Circle();             // allowed
Shape myShape2  = new Rectangle();        // allowed
Rectangle myRectangle = new Shame();      // NOT allowed
```

Polymorphism (continued)

- Can also declare a `class` final using the keyword `final`
- If a `class` is declared `final`, then no other class can be derived from this class
- Java does not use late binding for methods that are `private`, marked `final`, or `static`
  - `Why not?`

Casting

- You cannot automatically make reference variable of subclass type point to object of its superclass
- Suppose that `supRef` is a reference variable of a superclass type and `supRef` points to an object of its subclass:
  - Can use a cast operator on `supRef` and make a reference variable of the subclass point to the object
  - If `supRef` does not point to a subclass object and you use a cast operator on `supRef` to make a reference variable of the subclass point to the object, then Java will throw a `ClassCastException`—indicating that the class cast is not allowed

Polymorphism (continued)

- Operator `instanceof`: determines whether a reference variable that points to an object is of a particular class type
- This expression evaluates to `true` if `p` points to an object of the `class BoxShape`; otherwise it evaluates to `false`:

```
p instanceof BoxShape
```

Static Binding

- The binding which can be resolved at compile time by compiler is known as static or early binding.
- Binding of all the static, private and final methods is done at compile-time .
- Advantages
    - Static binding is better performance wise (no extra overhead is required).
    - Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class.
    - Hence compiler doesn’t have any difficulty to determine object of class (local class for sure).

Static Binding Example

```
public class NewClass {
    public static class superclass {
        static void print()
        {
            System.out.println("print in superclass.");
        }
    }
    public static class subclass extends superclass {
        static void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

Output:

```
print in superclass.
print in superclass.
```

Dynamic Binding

- In Dynamic binding (also called as Late Binding) compiler doesn’t decide the method to be called.
- Overriding is a perfect example of dynamic binding.
- In overriding both parent and child classes have same method .

Dynamic Binding Example

```
public class NewClass {
    public static class superclass {
        void print()
        {           System.out.println("print in superclass.");       }
    }
      public static class subclass extends superclass {
        @Override
        void print()        {
            System.out.println("print in subclass.");        }
    }
      public static void main(String[] args)
    {        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();    }                    }
Output:
print in superclass.
print in subclass.
```

Dynamic Binding Example

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Output:
```
dog is eating...
```

Abstract Methods

- A method that has only the heading with no body
- Must be implemented in a subclass
- Must be declared abstract

```
public double abstract area();
public void abstract print();
public abstract object larger(object,
                                    object);
void abstract insert(int insertItem);
```

Abstract Classes

- A class that is declared with the reserved word `abstract` in its heading
- An abstract class can contain instance variables, constructors, finalizers, and non-abstract methods
- An abstract class can contain abstract methods

Abstract Classes (Contd.)

- If a class contains an abstract method, the class must be declared abstract
- You cannot instantiate an object of an abstract class type; can only declare a reference variable of an abstract class type
- You can instantiate an object of a subclass of an abstract class, but only if the subclass gives the definitions of *all* the abstract methods of the superclass

Abstract Class Example

```
public abstract class AbstractClassExample
{
    protected int x;
    public void abstract print();

    public void setX(int a)
    {
         x = a;
    }

    public AbstractClassExample()
    {
        x = 0;
    }
}
```

Abstract Class Example(contd)

```
public class Abs1 extends AbstractClassExample{
   Public Abs1(){
        super():}
   public void print(){
        System.out.println(this.X);}
   public static void main(String[] args){
  //AbstractClassExample a1 = new AbstractClassExample();
  //Compile time error since can't instantiate abstract class
   Abs1 a2=new Abs1();
   a2. print();
   a2.setX(5);
   a2. print();     }    }
```

OUTPUT:

```
0
5
```

Single vs. Multiple Inheritance

- Some object-oriented languages allow *multiple inheritance,* which allows a class to be derived from two or more classes, inheriting the members of all parents
- The price: collisions, such as the same variable name, same method name in two parents, have to be resolved
- Java decision: *single inheritance*, meaning that a derived class can have only one parent class

Java Interface

- A Java *interface* is a collection of constants and abstract methods
  - abstract method: a method header without a method body; we declare an abstract method using the modifier `abstract`
  - since all methods in an interface are abstract, the `abstract` modifier is usually left off
- A class that contains only abstract methods and/or named constants
- How Java implements **multiple inheritance**
- To be able to handle a variety of events, **Java allows a class to implement more than one interface**
- **Methods** in an interface have **public** visibility by **default**
- Pass method descriptions, not implementation
- This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Classes ***implements*** interfaces

Interfaces Definition

- Syntax (appears like abstract class):
- Example:

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration – only method body
}
```

```
interface Speaker {
    public void speak( );
}
```

49

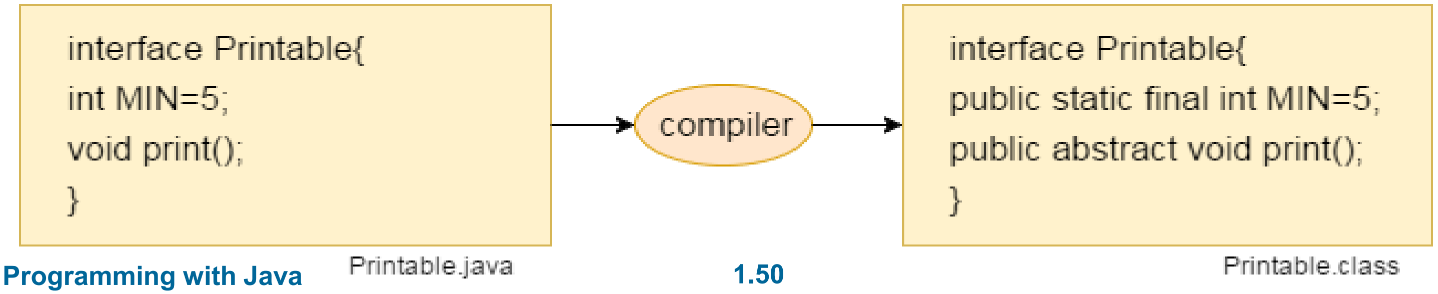
Interface: Syntax

interface is a reserved word

```
public interface Doable
{
    public static final String NAME;
    public void doThis();
    public int doThat();
    public void doThis2 (float value,
char ch);
   public boolean doTheOther(int num);
}
```

A semicolon immediately follows each method header

No method in an interface has a definition (body)



```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

Implementing an Interface

- A class formally implements an interface by
  - stating so in the class header in the `implements` clause
  - a class can implement multiple interfaces: the interfaces are listed in the implements clause, separated by commas
- If a class asserts that it implements an interface, it must define all methods in the interface or the compiler will produce errors
- Why use?
  - It is used to achieve abstraction.
  - By interface, we can support the functionality of multiple inheritance.
  - It can be used to achieve loose coupling.

Implementing Interfaces

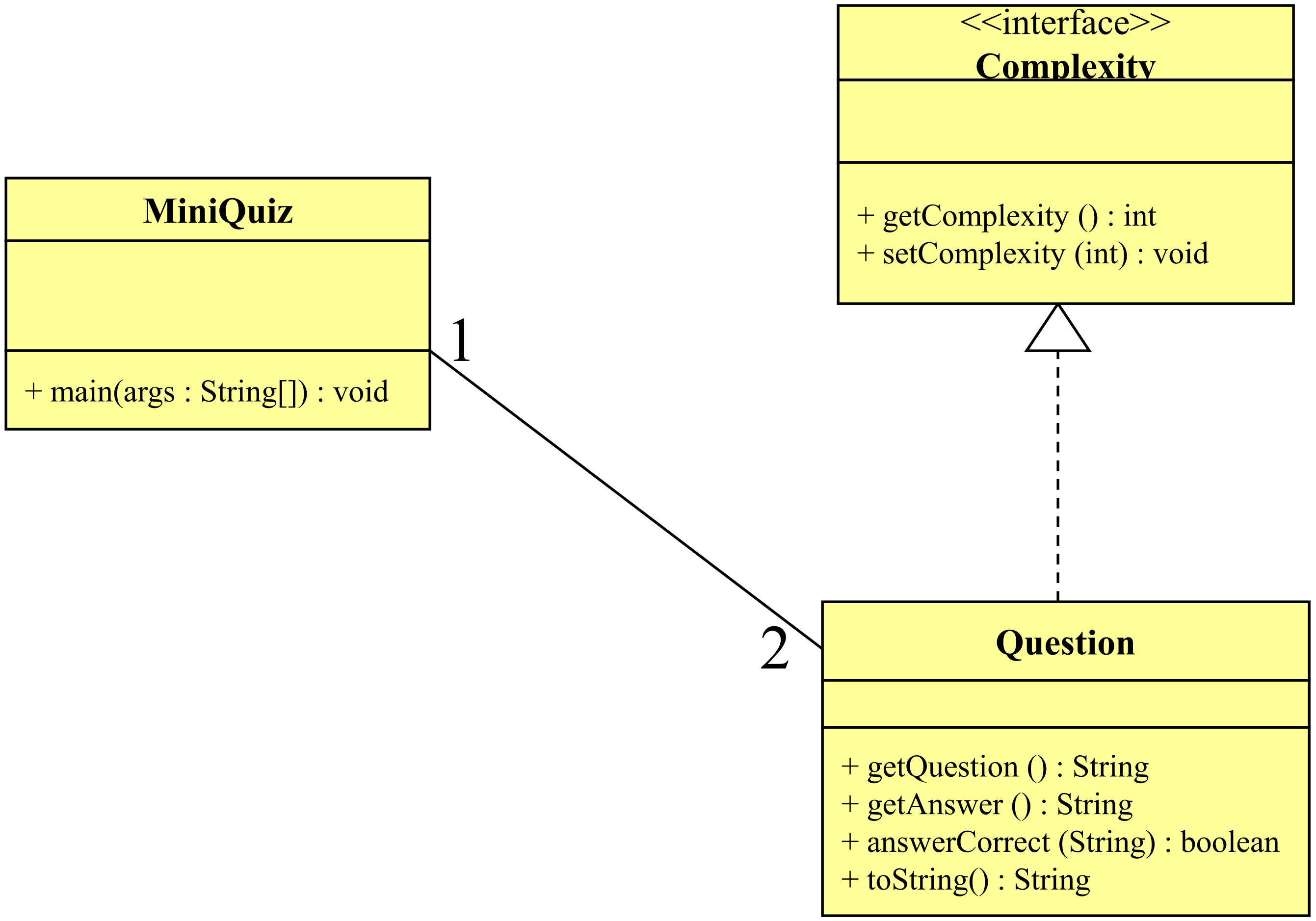
```
public class Something implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

```
public class ManyThings implements Doable, AnotherDoable
```

Interfaces: An ExampleUML Diagram



<<interface>>
Complexity
+ getComplexity () : int
+ setComplexity (int) : void
MiniQuiz
+ main(args : String[]) : void
1
2
Question
+ getQuestion () : String
+ getAnswer () : String
+ answerCorrect (String) : boolean
+ toString() : String

Implementing Interfaces

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, ...]
{
    // Body of Class
}
```

Implementing Interfaces Example

```
class  Politician implements Speaker {
        public void speak(){
            System.out.println("Talk politics");
        }
}
```

```
class  Priest implements Speaker {
        public void speak(){
            System.out.println("Religious Talks");
        }
}
```

```
class  Lecturer implements Speaker {
        public void speak(){
            System.out.println("Talks JAVA!");
        }
}
```

Extending Interfaces

- Like classes, **interfaces can also extend another interface**. The new sub-interface will inherit all the members of the superinterface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends
InterfaceName1 {
    // Body of InterfaceName2
}
```

Inheritance and Interface Implementation

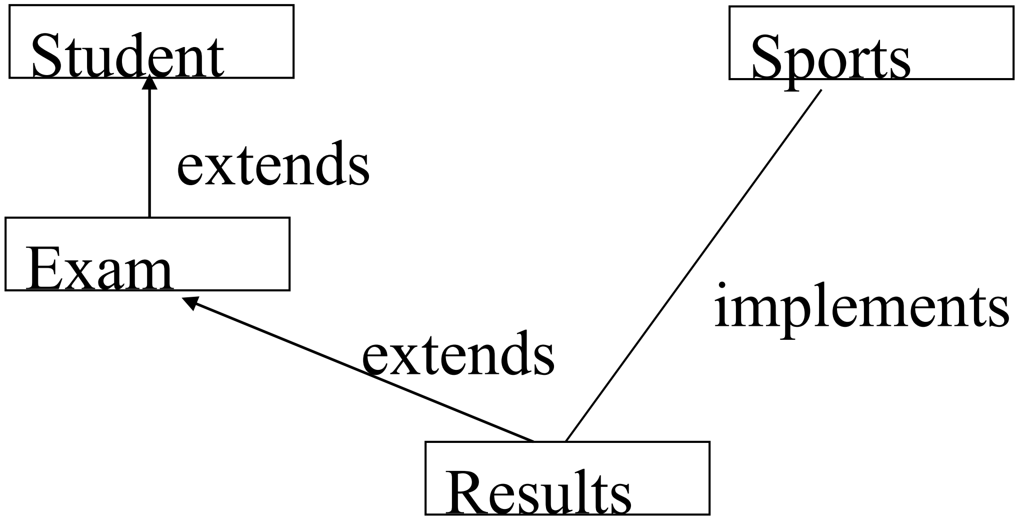
- A general form of interface implementation:

```
class ClassName extends SuperClass implements InterfaceName [,
InterfaceName2, ...]
{
    // Body of Class
}
```

- This shows a class can extended another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Student Assessment Example

- Consider a university where students who participate in the national games or Olympics are given some grace marks. Therefore, the final marks awarded = Exam_Marks + Sports_Grace_Marks. A class diagram representing this scenario is as follow:



Software Implementation

```
class Student
{
    // student no and access methods
}
interface Sport
{
    // sports grace marks (say 5 marks) and abstract methods
}
class Exam extends Student
{
    // example marks (test1 and test 2 marks) and access methods
}
class Results extends Exam implements Sport
{
    // implementation of abstract methods of Sport interface
    // other methods to compute total marks = test1+test2+sports_grace_marks;
    // other display or final results access methods
}
```

Interfaces: Examples from Java Standard Class Library

- The Java Standard Class library defines many interfaces:
  - the `Iterator` interface contains methods that allow the user to move through a collection of objects easily
    - `hasNext(), next(), remove()`
  - the `Comparable` interface contains an abstract method called `compareTo`, which is used to compare two objects

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

Polymorphism via Interfaces

- Define a polymorphism reference through interface
  - declare a reference variable of an interface type

```
Doable obj;
```

  - the `obj` reference can be used to point to any object of any class that implements the `Doable` interface

  - the version of `doThis` depends on the type of object that `obj` is referring to:

```
obj.doThis();
```

Examples

```
Speaker guest;
guest = new Philosopher();
guest.speak();

guest = Dog();
guest.speak();
```

```
Speaker special;
special = new Philosopher();

special.pontificate(); // compiler error
```

```
Speaker special;
special = new Philosopher();

((Philosopher)special).pontificate();
```

```
public interface Speaker
{
   public void speak();
}

class Philosopher extends Human
   implements Speaker
{
   //
   public void speak()
    {…}
   public void pontificate()
   {…}
}
class Dog extends Animal
   implements Speaker
{
   //
   public void speak()
    {
   …    }}
```

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface **inherits all abstract methods of the parent**
- A class implementing the child interface must define all methods from both the parent and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

Java Lambda Expressions

- Lambda expression is a new and important feature of Java which was included in Java SE 8.
- It provides a clear and concise way to represent one method interface using an expression.
  - It is very useful in collection library.
  - It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface.
- It saves a lot of code.
- In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- **Java lambda expression is treated as a function, so compiler does not create .class file.**

Functional Interface

- Lambda expression provides implementation of *functional interface*.
- **An interface which has only one abstract method is called functional interface.**
- Java provides an anotation *@FunctionalInterface*, which is used to declare an interface as functional interface.
- **Why use Lambda Expression**
  1. To provide the implementation of Functional interface.
  2. Less coding.

Java Lambda Expression Syntax

(argument-list) -> {body}

- Java lambda expression is consisted of three components.
  1) **Argument-list:** It can be empty or non-empty as well.
  2) **Arrow-token:** It is used to link arguments-list and body of expression.
  3) **Body:** It contains expressions and statements for lambda expression.
- **No Parameter Syntax**

```
() -> {  //Body of no parameter lambda  }
```

- **One Parameter Syntax**

```
(p1) -> {  //Body of single parameter lambda  }
```

- **Two Parameter Syntax**

```
(p1,p2) -> {  //Body of multiple parameter lambda  }
```

Example (Without Lambda Expression)

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;
          //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Output:

```
Drawing 10
```

Example (With Lambda Expression)

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Output:

```
Drawing 10
```

Java Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
        public static void main(String[] args) {
                Sayable s=()->{
                        return "I have nothing to say.";
                };
        System.out.println(s.say());
        }
}
```

Output:

```
I have nothing to say.
```

Java Lambda Expression Example: Single Parameter

```
interface Sayable{
       public String say(String name);
   }
   public class LambdaExpressionExample4{
       public static void main(String[] args) {
            // Lambda expression with single parameter.
           Sayable s1=(name)->{
               return “I LOVE "+name;
           };
           System.out.println(s1.say(“JAVA"));
           // You can omit function parentheses
           Sayable s2= name ->{
               return “I LOVE"+name;
           };
           System.out.println(s2.say(“JAVA"));
       }
   }
```

Output:

```
I LOVE JAVA
I LOVE JAVA
```

Java Lambda Expression Example: Multiple Parameter

```
interface Addable{
      int add(int a,int b);
  }
    public class LambdaExpressionExample5{
      public static void main(String[] args) {
          // Multiple parameters in lambda expression
          Addable ad1=(a,b)->(a+b);
          System.out.println(ad1.add(10,20));
          // Multiple parameters with data type in lambda expression
          Addable ad2=(int a,int b)->(a+b);
          System.out.println(ad2.add(100,200));
      }
  }
```

Output:

```
30
300
```

Java Lambda Expression Example: with or without return keyword

```
interface Addable{
       int add(int a,int b);
   }
    public class LambdaExpressionExample6 {
       public static void main(String[] args) {
           // Lambda expression without return keyword.
           Addable ad1=(a,b)->(a+b);
           System.out.println(ad1.add(10,20));
           // Lambda expression with return keyword.
           Addable ad2=(int a,int b)->{
                               return (a+b);
                               };
           System.out.println(ad2.add(100,200));
       }
   }
```

Output:

```
30
300
```

Java Lambda Expression Example: Foreach Loop

```
import java.util.*;
 public class LambdaExpressionExample7{
     public static void main(String[] args) {
         List<String> list=new ArrayList<String>();
         list.add("ankit");
         list.add("mayank");
         list.add("irfan");
         list.add("jai");
          list.forEach(
             (n)->System.out.println(n)
         );
     }
 }
```

Output:

```
ankit
mayank
irfan
Jai
```

Java Lambda Expression Example: Multiple Statements

```
@FunctionalInterface

   interface Sayable{
       String say(String message);
   }
public class LambdaExpressionExample8{
       public static void main(String[] args) {
          // You can pass multiple statements in lambda expression
           Sayable person = (message)-> {
               String str1 = "I would like to say, ";
               String str2 = str1 + message;
               return str2;
           };
               System.out.println(person.say("time is precious."));
       }
   }
```

Output:

```
I would like to say, time is precious.
```

Use of Method References

- You use lambda expressions to create anonymous methods.
- Sometimes, however, a lambda expression does nothing but call an existing method.
- In those cases, it's often clearer to refer to the existing method by name.
- Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.
  - // To refer a method in an object

  Object :: methodName

  - // To print all element in a list

  list.forEach(s -> System.out.println(s));

  - // Shorthand to print all element in a list

  list.forEach(System.out::println);

  use method reference rather than using a single function lambda expression

Kinds of Method References

Kind	Syntax	Examples
Reference to a **static method**	ContainingClass::staticMethodName	Person::compareByAge MethodReferencesExamples::appendStrings
Reference to an **instance method of a particular object**	containingObject::instanceMethodName	myComparisonProvider::compareByName myApp::appendStrings2
Reference to an **instance method of an arbitrary object of a particular type**	ContainingType::methodName	String::compareToIgnoreCase String::concat
Reference to a **constructor**	ClassName::new	HashSet::new

Reference to a Static Method

- **If a Lambda expression is like:**

  // If a lambda expression just call a static method of a class

  (args) -> Class.staticMethod(args)

- **Then method reference is like:**

  // Shorthand if a lambda expression just call a static method of a class

  Class::staticMethod

- **Capitalizing and printing a list of Strings:**

List<String> messages = Arrays.asList("hello", "baeldung", "readers!");

- We can achieve this by leveraging a simple lambda expression calling the StringUtils.capitalize() method directly:

  messages.forEach(word -> StringUtils.capitalize(word));

- Or, we can use a method reference to simply refer to the capitalize static method:

  messages.forEach(StringUtils::capitalize);

- **Notice that method references always utilize the :: operator**

Reference to an Instance Method of a Particular Object

- // If a lambda expression just call a default method of an object

```
(args) -> obj.instanceMethod(args)
```

Then method reference is like:

- // Shorthand if a lambda expression just call a default method of an object

```
obj::instanceMethod
```

Reference to an Instance Method of a Particular Object

```
public class Bicycle {
    private String brand;
    private Integer frameSize;
    // standard constructor, getters and setters                }
public class BicycleComparator implements Comparator {
    @Override
    public int compare(Bicycle a, Bicycle b) {
        return a.getFrameSize().compareTo(b.getFrameSize());    }           }
```

- Create a BicycleComparator object to compare bicycle frame sizes:

```
BicycleComparator bikeFrameSizeComparator = new BicycleComparator();
```

- We could use a lambda expression to sort bicycles by frame size, but we'd need to specify two bikes for comparison:

```
createBicyclesList().stream().sorted((a, b) -> bikeFrameSizeComparator.compare(a, b));
```

- Instead, we can use a method reference to have the compiler handle parameter passing for us:

```
createBicyclesList().stream().sorted(bikeFrameSizeComparator::compare);
```

Reference to an Instance Method of an Arbitrary Object of a Particular Type

Let's create an Integer list that we want to sort:

```
List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);
```

If we use a classic lambda expression, both parameters need to be explicitly passed,

```
numbers.stream().sorted((a, b) -> a.compareTo(b));
```

while using a method reference is much more straightforward:

```
numbers.stream().sorted(Integer::compareTo);
```

Example Program

```
import java.util.function.BiFunction;
public class MethodReferencesExamples {
        public static <T> T mergeThings(T a, T b, BiFunction<T, T, T> merger) {
        return merger.apply(a, b);    }
        public static String appendStrings(String a, String b) {
        return a + b;    }
        public String appendStrings2(String a, String b) {
        return a + b;    }
    public static void main(String[] args) {
       MethodReferencesExamples myApp = new MethodReferencesExamples();
        // Calling the method mergeThings with a lambda expression
        System.out.println(MethodReferencesExamples.mergeThings("Hello ", "World!", (a, b) -> a + b));
         // Reference to a static method
       System.out.println(MethodReferencesExamples.mergeThings("Hello ", "World!", MethodReferencesExamples::appendStrings));
        // Reference to an instance method of a particular object
       System.out.println(MethodReferencesExamples.mergeThings("Hello ", "World!", myApp::appendStrings2));
       // Reference to an instance method of an arbitrary object of a particular type
        System.out.println(MethodReferencesExamples.
            mergeThings("Hello ", "World!", String::concat));    }                }
```

All the System.out.println() statements print the same thing: Hello World!

Reference to a Constructor

- We can reference a constructor in the same way that we referenced a static method in our first example. The only difference is that we'll use the new keyword.
- Let's create a Bicycle array out of a String list with different brands:

```
List<String> bikeBrands = Arrays.asList("Giant", "Scott", "Trek", "GT");
```

First, we'll add a new constructor to our Bicycle class:

```
public Bicycle(String brand) {
        this.brand = brand;
        this.frameSize = 0;  }
```

Next, we'll use our new constructor from a method reference and make a Bicycle array from the original String list:

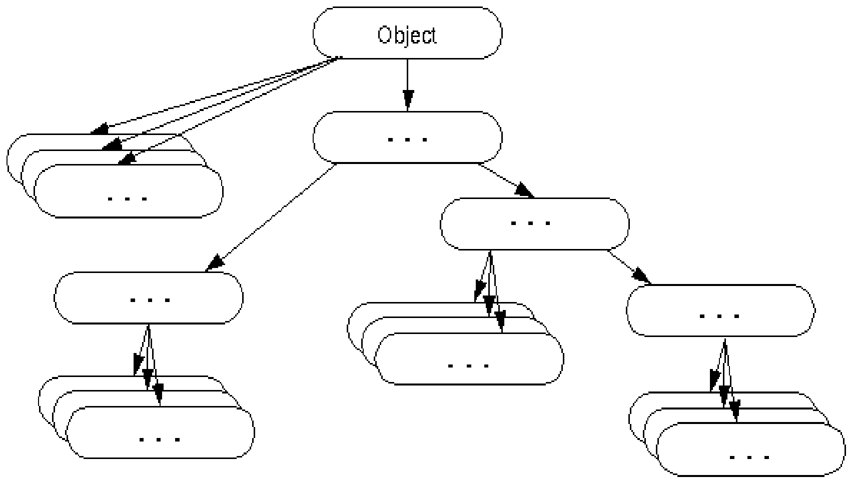
```
bikeBrands.stream().map(Bicycle::new).toArray(Bicycle[]::new);
```

Object Class

- The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.
- Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

```
Object obj=getObject();
```

- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Methods of Object class

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

toString()

- toString() provides String representation of an Object and used to **convert an object to String.**
- The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@’, and the unsigned hexadecimal representation of the hash code of the object.
- It is always recommended to override **toString()** method to get our own String representation of Object.

```
Student s = new Student();
// Below two statements are equivalent
System.out.println(s);
System.out.println(s.toString());
```

hashCode()

- For every object, **JVM generates a unique number** which is hashcode.
- It returns **distinct integers for distinct objects.**
- A common misconception about this method is that **hashCode() method returns the address of object, which is not correct.**
- It convert the internal address of object to an integer by using an algorithm.
- The hashCode() method is **native** because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.
  - Returns a hash value that is used to search object in a collection

Example

```
public class Student{
    static int last_roll = 100;
    int roll_no;
    Student()    {             // Constructor
             roll_no = last_roll;
             last_roll++; }
    @Override      // Overriding hashCode()
    public int hashCode(){
            return roll_no;    }
    public static void main(String args[]){    // Driver code
         Student s1 = new Student();
         Student s2 = new Student();
         // Below two statements are equivalent
        System.out.println(s1);
        System.out.println(s1.toString());
        System.out.println(s1.roll_no);
        System.out.println(s2);
        System.out.println(s2.toString());
        System.out.println(s2.roll_no);    } }
```

Output :

```
Student@64
Student@64
100
Student@65
Student@65
101
```

equals(Object obj)

- Compares the given object to “this” object (the object on which the method is called).
- It gives a generic way to compare objects for equality.
- It is recommended to override **equals(Object obj)** method to get our own equality condition on Objects.

Note : It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

getClass()

- Returns the class object of “this” object and used to get actual runtime class of the object. It can also be used to get metadata of this class.
- The returned Class object is the object that is locked by static synchronized methods of the represented class.
- As it is final so we don’t override it.

```
public class Test{
          public static void main(String[] args)    {
                    Object obj = new String("GeeksForGeeks");
                    Class c = obj.getClass();
                    System.out.println("Class of Object obj is : "+
c.getName());  }
}
```

Output:

```
Class of Object obj is : java.lang.String
```

Note :After loading a .class file, JVM will create an object of the type java.lang.Class in the Heap area. We can use this class object to get Class level information. It is widely used in Reflection

finalize()

- This method is called just before an object is garbage collected.
- It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object.
- We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks.
- For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.

Note: finalize method is called just once on an object even though that object is eligible for garbage collection multiple times.

Example

```
public class Test{
    public static void main(String[] args)    {
        Test t = new Test();
        System.out.println(t.hashCode());
         t = null;
         // calling garbage collector
        System.out.println("end");    }
     @Override
    protected void finalize()
    {        System.out.println("finalize method called");    }}
```

Output:

```
366712642
finalize method called
end
```

Other Methods

- **clone()** : It returns a new object that is exactly the same as this object.
- **wait()**, **notify() notifyAll():** related to Concurrency in Inter thread communication in java

Object wrappers

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.
- Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically.
- The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.
  - **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
  - **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
  - **Synchronization:** Java synchronization works with objects in Multithreading.
  - **java.util package:** The java.util package provides the utility classes to deal with objects.
  - **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Wrapper classes

- The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing,
- For example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
 public class WrapperExample1{
   public static void main(String args[]){
   //Converting int into Integer
   int a=20;
   Integer i=Integer.valueOf(a);//converting int into Integer explicitly
   Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
       System.out.println(a+" "+i+" "+j);
   }
}
```

Output:

```
20 20 20
```

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.
- It is the reverse process of autoboxing.
- Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
 public class WrapperExample2{
   public static void main(String args[]){
   //Converting Integer to int
   Integer a=new Integer(3);
   int i=a.intValue();//converting Integer to int explicitly
   int j=a;//unboxing, now compiler will write a.intValue() internally

   System.out.println(a+" "+i+" "+j);
   }
}
```

Output:

```
3 3 3
```

Java Wrapper classes Example

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;
  //Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;
```

Java Wrapper classes Example (Contd.)

```
  //Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);
  //Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;
```

Java Wrapper classes Example (Contd.)

```
//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}}
```

Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
```

Java Wrapper classes Example (Contd.)

Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

```
 class Javatpoint{
    private int i;
    Javatpoint(){}
    Javatpoint(int i){
    this.i=i;     }
    public int getValue(){
    return i;     }
    public void setValue(int i){
    this.i=i;     }
    @Override
    public String toString() {
      return Integer.toString(i);     }     }
    //Testing the custom wrapper class
    public class TestJavatpoint{
    public static void main(String[] args){
    Javatpoint j=new Javatpoint(10);
    System.out.println(j);     }}
```

Output:

```
10
```

Reflection

- Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.
- The required classes for reflection are provided under java.lang.reflect package.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.
- Reflection can be used to get information about –
  1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
  2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
  3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

Example

```
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;
// class whose object is to be created
class Test{
    // creating a private field
    private String s;
      // creating a public constructor
    public Test()  {  s = "GeeksforGeeks"; }
    // Creating a public method with no arguments
    public void method1()  {
        System.out.println("The string is " + s);    }
      // Creating a public method with int as argument
    public void method2(int n)  {
        System.out.println("The number is " + n);    }
      // creating a private method
    private void method3() {
        System.out.println("Private method invoked");    }}
  class Demo{
 public static void main(String args[]) throws Exception    {
        // Creating object whose property is to be checked
        Test obj = new Test();
```

Example (Contd)

```
// Creating class object from the object using
        // getclass method
        Class cls = obj.getClass();
        System.out.println("The name of class is " +  cls.getName());
          // Getting the constructor of the class through the
        // object of the class
        Constructor constructor = cls.getConstructor();
        System.out.println("The name of constructor is " + constructor.getName());
          System.out.println("The public methods of class are : ");
          // Getting methods of the class through the object
        // of the class by using getMethods
        Method[] methods = cls.getMethods();
          // Printing method names
        for (Method method:methods)
            System.out.println(method.getName());
          // creates object of desired method by providing the
        // method name and parameter class as arguments to
        // the getDeclaredMethod
        Method methodcall1 = cls.getDeclaredMethod("method2", int.class);
          // invokes the method at runtime
        methodcall1.invoke(obj, 19);
```

Example (Contd)

```
// creates object of the desired field by providing
 // the name of field as argument to the
 // getDeclaredField method
 Field field = cls.getDeclaredField("s");
   // allows the object to access the field irrespective
 // of the access specifier used with the field
 field.setAccessible(true);
   // takes object and the new value to be assigned
 // to the field as arguments
 field.set(obj, "JAVA");
  // Creates object of desired method by providing the
 // method name as argument to the getDeclaredMethod
 Method methodcall2 = cls.getDeclaredMethod("method1");
   // invokes the method at runtime
 methodcall2.invoke(obj);
  // Creates object of the desired method by providing
 // the name of method as argument to the
 // getDeclaredMethod method
 Method methodcall3 = cls.getDeclaredMethod("method3");
   // allows the object to access the method irrespective
 // of the access specifier used with the method
 methodcall3.setAccessible(true);
   // invokes the method at runtime
 methodcall3.invoke(obj);    }              }
```

Example (Contd)

Output :

```
The name of class is Test
The name of constructor is Test
The public methods of class are :
method2
method1
wait
wait
wait
equals
toString
hashCode
getClass
notify
notifyAll
The number is 19
The string is JAVA
Private method invoked
```

Notes to Remember about Reflections

1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
   - getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is

     Class.getDeclaredMethod(name, parametertype)

     name- the name of method whose object is to be created

     parametertype - parameter is an array of Class objects
   - invoke() : To invoke a method of the class at runtime we use following method–

     Method.invoke(Object, parameter)

     If the method of the class doesn't accepts any parameter then null is passed as argument.

2. Through reflection we can access the private variables and methods of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
   - Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.
   - Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

Reflection

- **Advantages of Using Reflection:**
  - **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
  - **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.

- **Drawbacks:**
  - **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
  - **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

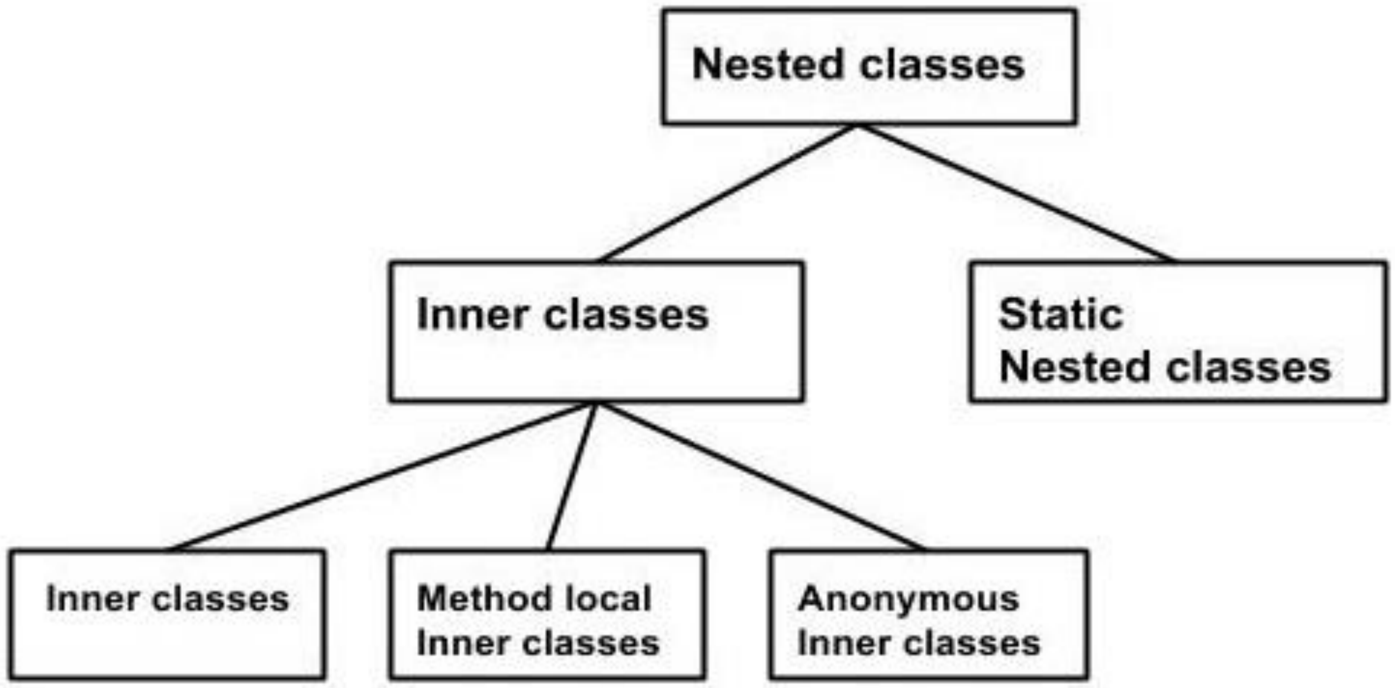
Java Inner Classes (Nested Classes)

- **Java inner class** or nested class is a class that is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.
- Additionally, it can access all the members of the outer class, including private data members and methods.
- **Syntax of Inner class**

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Nested Classes

- Nested classes are divided into two types −
  - **Non-static nested classes(Inner Classes)** − These are the non-static members of a class.
  - **Static nested classes** − These are the static members of a class.



Advantage of Java inner classes

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

Need of Java Inner class

- Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.
- If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Difference between nested class and inner class in Java

- An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

- There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.
- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

Inner Class Example

```
class OuterClass {
  int x = 10;
  class InnerClass {
    int y = 5;
  }
}
public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
// Outputs 15 (5 + 10)
```

Private Inner Class Example

- Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

```
class OuterClass {
  int x = 10;
  private class InnerClass {
    int y = 5;  }       }
public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);  }        }
```

If you try to access a private inner class from an outside class, an error occurs:

```
Main.java:13: error: OuterClass.InnerClass has private access in OuterClass
    OuterClass.InnerClass myInner = myOuter.new InnerClass();          ^
```

Static Inner Class Example

- An inner class can also be static, which means that you can access it without creating an object of the outer class:

```
class OuterClass {
  int x = 10;
  static class InnerClass {
    int y = 5;  }       }
public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}
// Outputs 5
```

just like static attributes and methods, a static inner class does not have access to members of the outer class

Access Outer Class From Inner Class

```
class OuterClass {
  int x = 10;
  class InnerClass {
    public int myInnerMethod() {
      return x;    }      }            }
public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```

Anonymous inner class example

```
abstract class AnonymousInner {
   public abstract void mymethod();
}
public class Outer_class {
   public static void main(String args[]) {
      AnonymousInner inner = new AnonymousInner() {
         public void mymethod() {
            System.out.println("This is an example of anonymous inner class");
         }
      };
      inner.mymethod();
   }
}
```

Output

```
This is an example of anonymous inner class
```

Method-local Inner Class

```
public class Outerclass {
   // instance method of the outer class
   void my_Method() {
      int num = 23;
      // method-local inner class
      class MethodInner_Demo {
         public void print() {
            System.out.println("This is method inner class "+num);         }         } // end of inner class
      // Accessing the inner class
      MethodInner_Demo inner = new MethodInner_Demo();
      inner.print();      }
      public static void main(String args[]) {
      Outerclass outer = new Outerclass();
      outer.my_Method();
   }
}
```

Output

```
This is method inner class 23
```

Errors - Intro

- Due to design errors or coding errors, our programs may fail in unexpected ways during execution. An exception is a condition that is caused by run time error in the program. The purpose of the exception handling mechanism is to provide a means to detect and report an “ecxceptional circumstances” .

Error

- An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. So it is our responsibility to detect and manage the error properly.

Types of error

- Runtime Errors: occur while the program is running if the environment detects an operation that is impossible to carry out.
- Logic Errors: occur when a program doesn't perform the way it was intended
- Syntax Errors: Arise because the rules of the language have not been followed. They are detected by the compiler.

Example of Run Time error

```
Class Error
{
public static void main(String args[])
 {
int a=10;
int b=5;
int c=5;

int x=a/(b+c);
System.out.println("x=" +x);
int y=a/(b-c);   // Errorr division by zero
System.out.println("y=" +y);
}
}
```

Errors and Error Handling

- Some typical causes of errors:
  - Memory errors (i.e. memory incorrectly allocated, memory leaks, “null pointer”)
  - File system errors (i.e. disk is full, disk has been removed)
  - Network errors (i.e. network is down, URL does not exist)
  - Calculation errors (i.e. divide by 0)
- More typical causes of errors:
  - Array errors (i.e. accessing element –1)
  - Conversion errors (i.e. convert ‘q’ to a number)
  - Can you think of some others?
- Exceptions – a better error handling
  - Exceptions are a mechanism that provides the best of both worlds.
  - Exceptions act similar to method return flags in that any method may raise and exception should it encounter an error.
  - Exceptions act like global error methods in that the exception mechanism is built into Java; exceptions are handled at many levels in a program, locally and/or globally.

Exception Handling

- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- An exception is an event that **occurs during the execution of a program**

  that disrupts the normal flow of instructions.
- ***Run time error*** *occurs during the execution of a program. In contrast, compile-time errors occur while a program is being compiled. Runtime errors indicate bugs in the program or problems that the designers had anticipated but could do nothing about. For example, running out of memory will often cause a runtime error.*

Difference between error & exception

- **Errors** indicate serious problems and abnormal conditions that most applications should not try to handle.
- Error defines problems that are not expected to be caught under normal circumstances by program. For example memory error, hardware error, JVM error etc.
  Exceptions are conditions within the code.
- A developer can handle such conditions and take necessary corrective actions. Few examples – DivideByZero exception, NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException

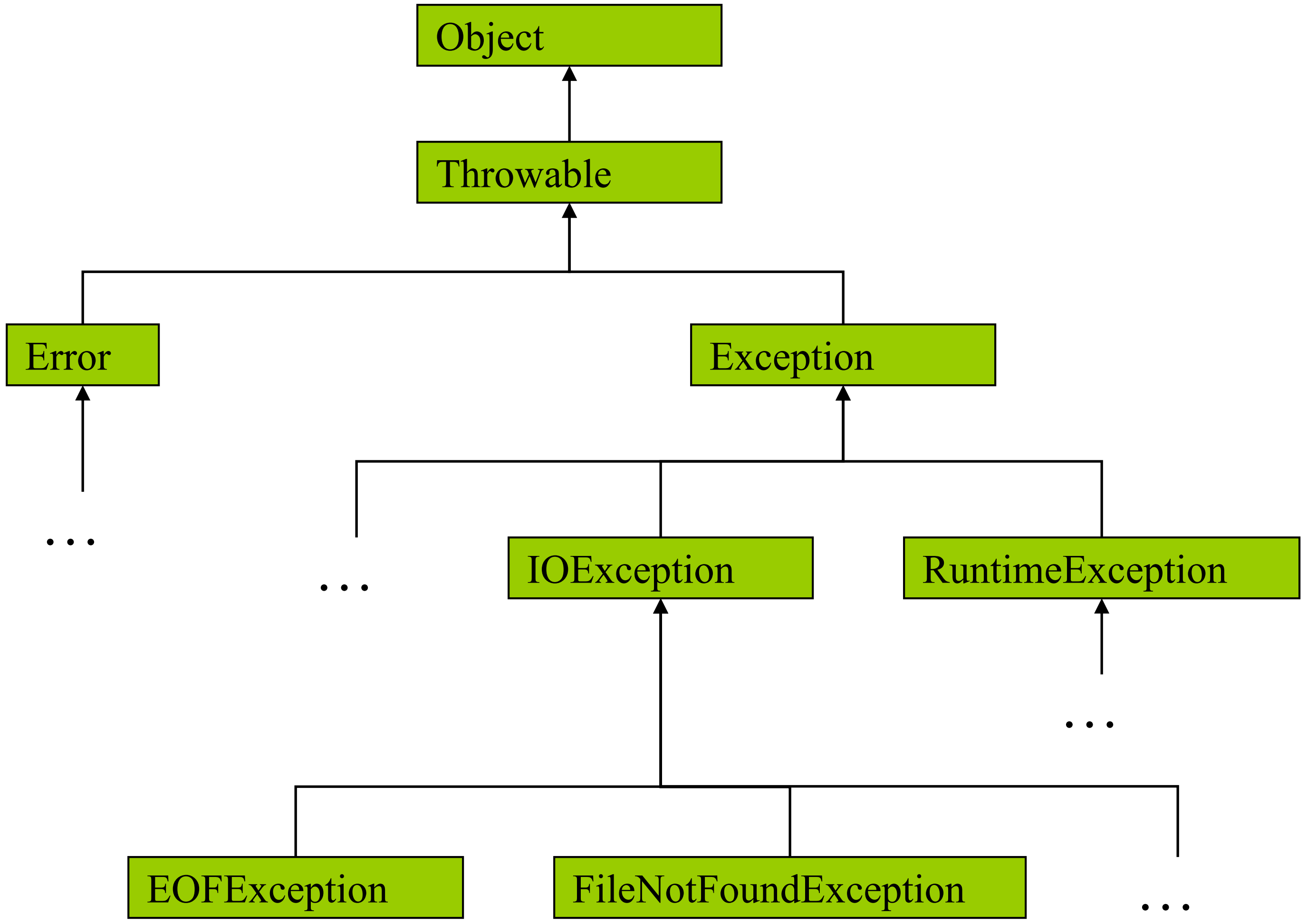
Handling Exceptions

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmers responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.
- Java allows you to create exception handlers.
- An *exception handler* is a section of code that gracefully responds to exceptions.
- The process of intercepting and responding to exceptions is called *exception handling*.
- The *default exception handler* deals with unhandled exceptions.
- The default exception handler prints an error message and crashes the program.

Exception Classes

- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.
- Classes that are derived from `Error`:
  - are for exceptions that are thrown when critical errors occur. (i.e.)
    - an internal error in the Java Virtual Machine, or
    - running out of memory.
- Applications should not try to handle these errors because they are the result of a serious condition.
- Programmers should handle the exceptions that are instances of classes that are derived from the `Exception` class.

Exception Classes



Exceptions

- How do you handle exceptions?
  - To handle the exception, you write a "try-catch" block. To pass the exception "up the chain", you declare a throws clause in your method or class declaration.
  - If the method contains code that may cause a checked exception, you MUST handle the exception OR pass the exception to the parent class (remember, every class has Object as the ultimate parent)
- To handle an exception, you use a *try* statement.

```
try
 {   (try block statements...)   }
 catch (ExceptionType ParameterName)
 {   (catch block statements...)         }
```

- First the keyword `try` indicates a block of code will be attempted (the curly braces are required).
- This block of code is known as a *try block*.

Handling Exceptions

- A *try block* is:
  - one or more statements that are executed, and
  - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.
- A catch clause begins with the key word `catch`:

  ```
  catch (ExceptionType ParameterName)
  ```

  - `ExceptionType` is the name of an exception class and
  - `ParameterName` is a variable name which will reference the exception object if the code in the try block throws an exception.
- The code that immediately follows the catch clause is known as a *catch block* (the curly braces are required).
- The code in the catch block is executed if the try block throws an exception.

Handling Exceptions

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
{
   File file = new File ("MyFile.txt");
   Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
```

- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.

Handling Exceptions

- The parameter must be of a type that is compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the catch block.
- Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception.

Example

```
class error2
{
public static void main(String arg[])
{
int a=10;
 int b=5;
 int c=5;
 int x,y;
    try
   {
    x=a/(b-c);
   }
    catch(ArithmeticException e)
   {
   System.out.println(“Division by Zero”);
   }
   Y=a/(b-c);
   System.out.println(“y=“+y);
   }
   }
```

Polymorphic References To Exceptions

- When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause.
- Most exceptions are derived from the `Exception` class.
- A `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that is derived from the `Exception` class.

```
try
{    number = Integer.parseInt(str);   }
catch (Exception e)
{     System.out.println("The following error occurred: "                 + e.getMessage());      }
```

- The `Integer` class's `parseInt` method throws a `NumberFormatException` object.
- The `NumberFormatException` class is derived from the `Exception` class.

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible `catch` clause found.
- The `catch` clauses must be listed from most specific to most general.

Exception Handlers

- There can be many polymorphic catch clauses.
- A try statement may have only one `catch` clause for each specific type of exception.

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
   System.out.println(str + " is not a number.");
}
```

Exception Handlers

- The `NumberFormatException` class is derived from the `IllegalArgumentException` class.

```
try
{
   number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
   System.out.println(str + " is not a number.");
}
```

Exception Handlers

- The previous code could be rewritten to work, as follows, with no errors:

```
try
{
  number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
  System.out.println(str +
                  " is not a number.");
}
catch (IllegalArgumentException e) //OK
{
  System.out.println("Bad number format.");
}
```

The `finally` Clause

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the `catch` clauses.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
finally
{
   (finally block statements...)
}
```

The `finally` Clause

- The *finally block* is one or more statements,
  - that are always executed after the try block has executed and
  - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.
- It indicates:
  - the method that was executing when an exception occurred and
  - all of the methods that were called in order to execute that method.

* The Method `printStackTrace`

- Used to determine the order in which the methods were called and where the exception was handled
- Java keeps track of the sequence of method calls in a program. When an exception occurs in a method, you can invoke the method `printStackTrace()` (which comes courtesy of the `Throwable` class) to determine the order in which the methods were called and where the exception was handled.

The Method `printStackTrace` (continued)

```
import java.io.*;
public class PrintStackTraceExample1
{
    public static void main(String[] args)
    {
        try
        {
            methodA();
        }
        catch (Exception e)
        {
            System.out.println(e.toString()
                               + " caught in main");
            e.printStackTrace();
        }
    }
```

The Method `printStackTrace` (continued)

```
    public static void methodA() throws Exception
    {
        methodB();
    }
    public static void methodB() throws Exception
    {
        methodC();
    }
    public static void methodC() throws Exception
    {
        throw new Exception("Exception generated "
                                   + "in method C");
    }
}
```

The Method `printStackTrace`

- Sample Run

```
java.lang.Exception: Exception generated in method C caught in main
java.lang.Exception: Exception generated in method C
    at PrintStackTraceExample1.methodC (PrintStackTraceExample1.java:30)
    at PrintStackTraceExample1.methodB (PrintStackTraceExample1.java:25)
    at PrintStackTraceExample1.methodA (PrintStackTraceExample1.java:20)
    at PrintStackTraceExample1.main (PrintStackTraceExample1.java:9)
```

Note: Because the methods A and B do not handle the exception thrown by C, they (A and B) **must include the '`throws`' clause in their heading**. Note how in `PrintStackTraceExample2`, `methodA()` does *not* include a `Throws` clause.

Uncaught Exceptions

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
  - normal execution of that method stops, and
  - the JVM searches for a compatible exception handler inside the method.
- If there is no exception handler inside the method:
  - control of the program is passed to the previous method in the call stack.
  - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the `main` method:
  - the main method must either handle the exception, or
  - the program is halted and the default exception handler handles the exception.

Checked and Unchecked Exceptions

- There are two categories of exceptions:
  - unchecked
  - checked.
- *Unchecked exceptions* are those that are derived from the `Error` class or the `RuntimeException` class.
- Exceptions derived from `Error` are thrown when a critical error occurs, and should not be handled.
- `RuntimeException` serves as a superclass for exceptions that result from programming errors.
- These exceptions can be avoided with properly written code.
- Unchecked exceptions, in most cases, should not be handled.
- All exceptions that are *not* derived from `Error` or `RuntimeException` are *checked exceptions*.
- If the code in a method can throw a checked exception, the method:
  - must handle the exception, or
  - it must have a `throws` clause listed in the method header.
- The `throws` clause informs the compiler what exceptions can be thrown from a method.

Checked and Unchecked Exceptions

```
// This method will not compile!
public void displayFile(String name)
{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);
   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
     System.out.println(inputFile.nextLine());
   }
   // Close the file.
   inputFile.close();
}
```

Checked and Unchecked Exceptions

- The code in this method is capable of throwing checked exceptions.
- The keyword `throws` can be written at the end of the method header, followed by a list of the types of exceptions that the method can throw.

```
public void displayFile(String name)
      throws FileNotFoundException
```

Checked Exceptions vs. Unchecked Exceptions

<u>RuntimeException</u>, <u>Error</u> 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.

Throwing Exceptions

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception. ***`throw new ExceptionType(MessageString);`***
- The `throw` statement causes an exception object to be created and thrown.
- The *MessageString* argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

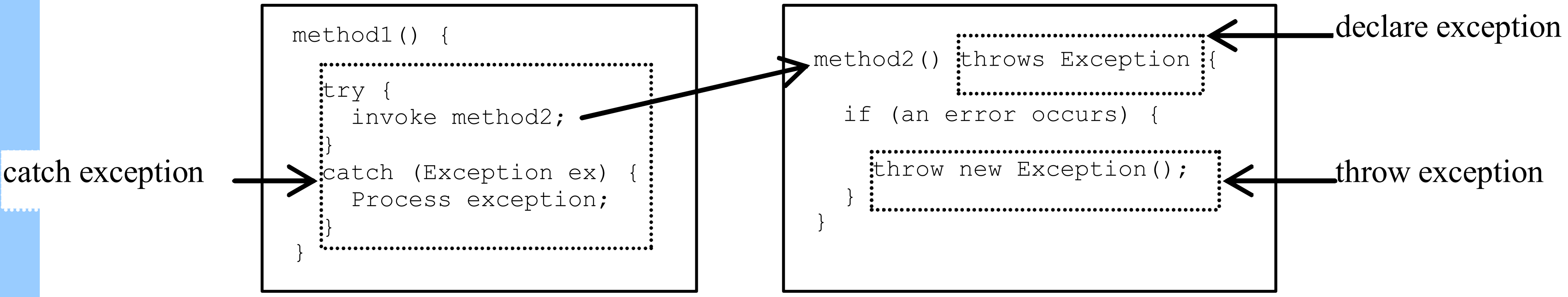
```
throw new Exception("Out of fuel");
```

- *Note: Don't confuse the `throw` statement with the `throws` clause.*

Creating Exception Classes

- You can create your own exception classes by deriving them from the `Exception` class or one of its derived classes.
- Some examples of exceptions that can affect a bank account:
  - A negative starting balance is passed to the constructor.
  - A negative interest rate is passed to the constructor.
  - A negative number is passed to the deposit method.
  - A negative number is passed to the withdraw method.
  - The amount passed to the withdraw method exceeds the account's balance.
- We can create exceptions that represent each of these error conditions.

Declaring, Throwing, and Catching Exceptions



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
   throws IOException

public void myMethod()
   throws IOException, OtherException
```

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();

TheException ex = new TheException();
throw ex;
```

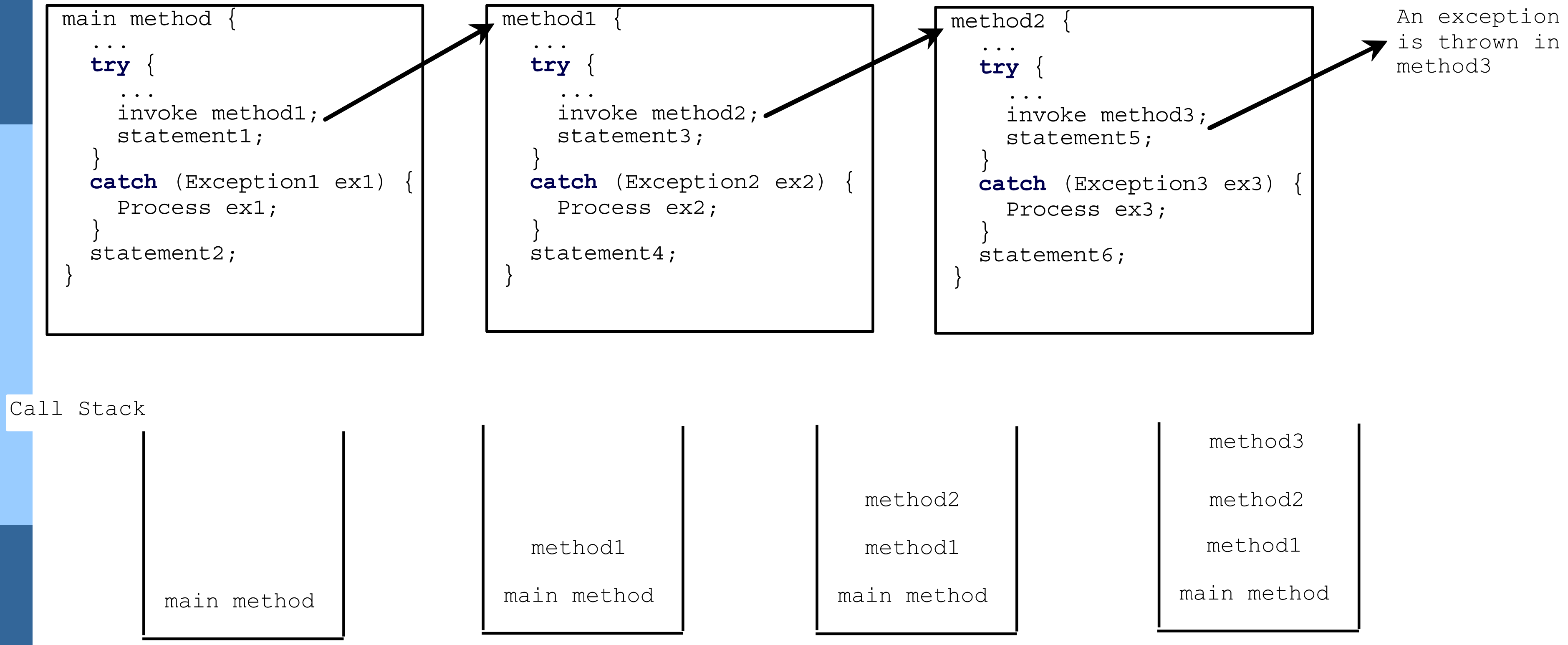
Throwing Exceptions Example

```
 /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

Catching Exceptions



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

Catch or Declare Checked Exceptions

Suppose p2 is defined as follows:

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

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 as shown in (a) or (b).

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

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

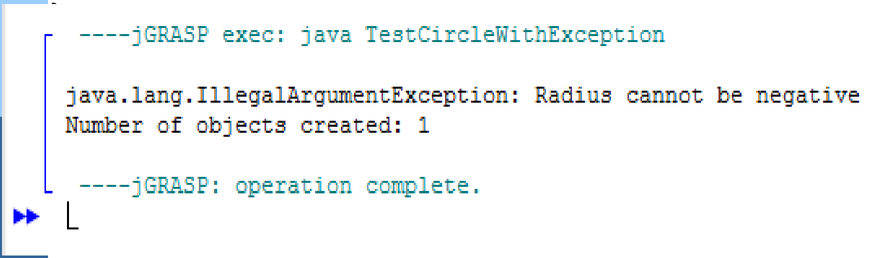
Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 8. The new setRadius method throws an exception if radius is negative.

TestCircleWithException   CircleWithException

TestCircleWithException

```
public class TestCircleWithException {
 public static void main(String[] args) {
   try {
     CircleWithException c1 = new
CircleWithException(5);
     CircleWithException c2 = new CircleWithException(-
5);
     CircleWithException c3 = new
CircleWithException(0);
   }
   catch (IllegalArgumentException ex) {
     System.out.println(ex);
   }

   System.out.println("Number of objects created: " +
     CircleWithException.getNumberOfObjects());
 }
```



```
 ----jGRASP exec: java TestCircleWithException

java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1

 ----jGRASP: operation complete.
```

CircleWithException

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

Rethrowing Exceptions

```
try {
   statements;
}
catch(TheException ex) {
   perform operations before exits;
   throw ex;
}
```

The `finally` Clause

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

animation

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

animation

Trace a Program Execution

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


Next statement;
```

The final block is always executed

animation

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

animation

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

animation

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

The exception is handled.

animation

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

The final block is always executed.

animation

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

animation

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
catch(Exception2 ex) {
   handling ex;
   throw ex;
}
finally {
   finalStatements;
}

Next statement;
```

Handling exception

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

When to Throw Exceptions

- An exception occurs in a method.
- **If you want the exception to be processed by its caller**,
  - Then **you should create an exception object and throw** it.
  - **If you can handle the exception in the method where it occurs, there is no need to throw it.**

When to Use Exceptions

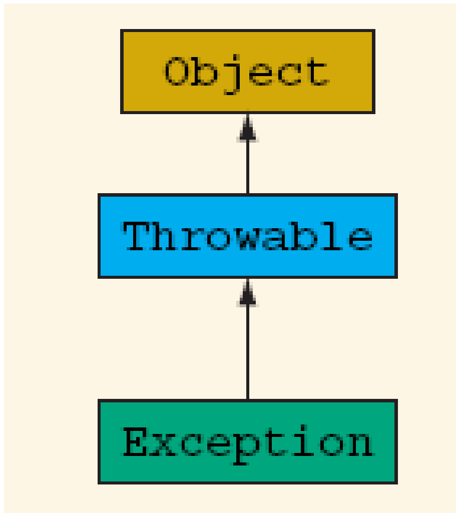
When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString()); }
catch (NullPointerException ex) {
  System.out.println("refVar is null");}
```

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Java Exception Hierarchy



It is not difficult to create your own exception classes by extending the class `Exception`. In the real world, most programmers rarely need to do so.

Java Exception Hierarchy (continued)

Constructors and Methods of the `class Throwable`

```
public Throwable()
   //Default constructor
   //Creates an instance of Throwable with an empty message string
```

```
public Throwable(String strMessage)
   //Constructor with parameters
   //Creates an instance of Throwable with message string
   //specified by the parameter strMessage
```

```
public String getMessage()
   //Returns the detailed message stored in the object
```

```
public void printStackTrace()
   //Method to print the stack trace showing the sequence of
   //method calls when an exception occurs
```

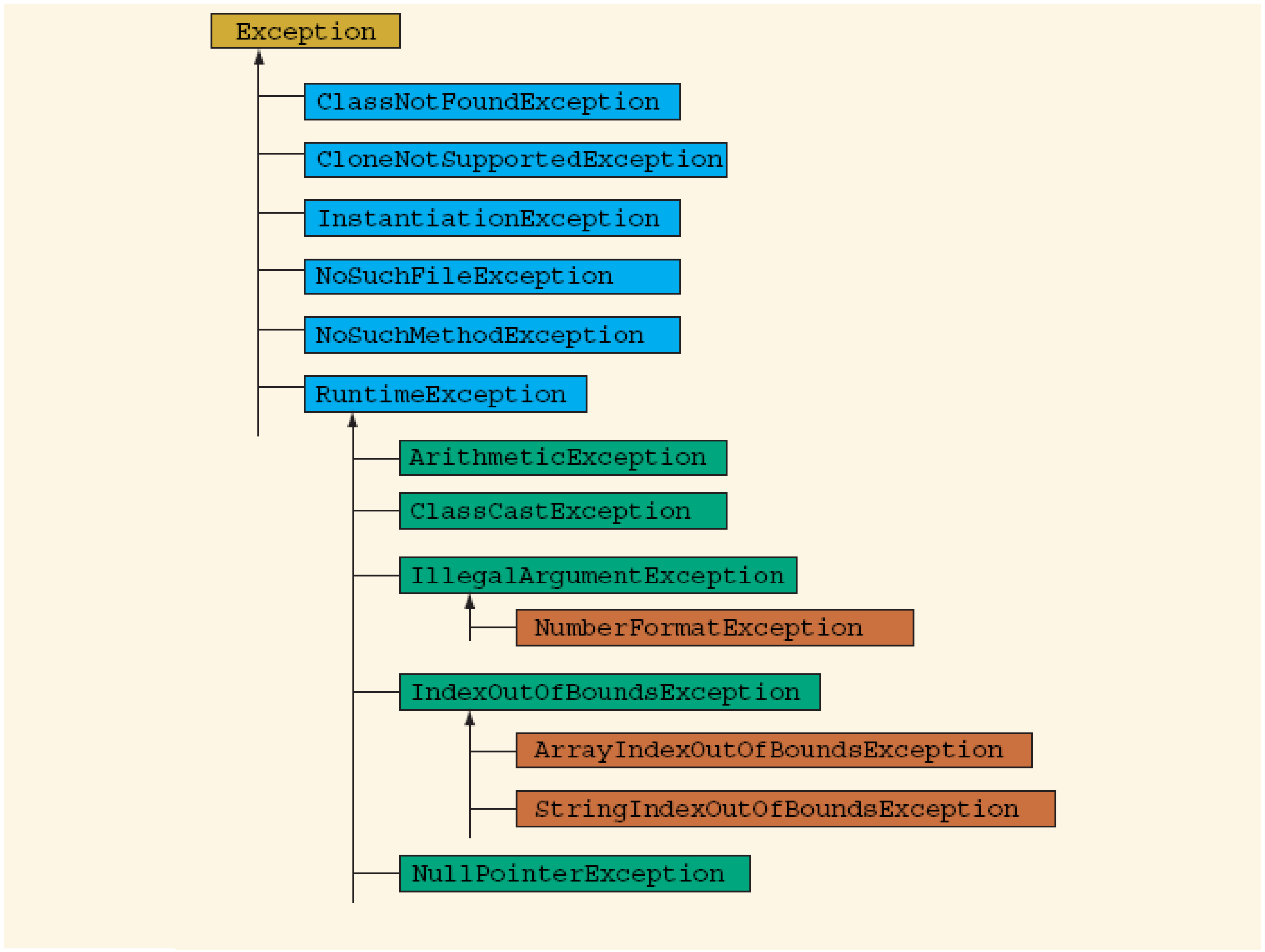
```
public void printStackTrace(PrintWriter stream)
   //Method to print the stack trace showing the sequence of
   //method calls when an exception occurs. Output is sent
   //to the stream specified by the parameter stream.
```

```
public String toString()
   //Returns a string representation of the Throwable object
```

Java’s Exception Class

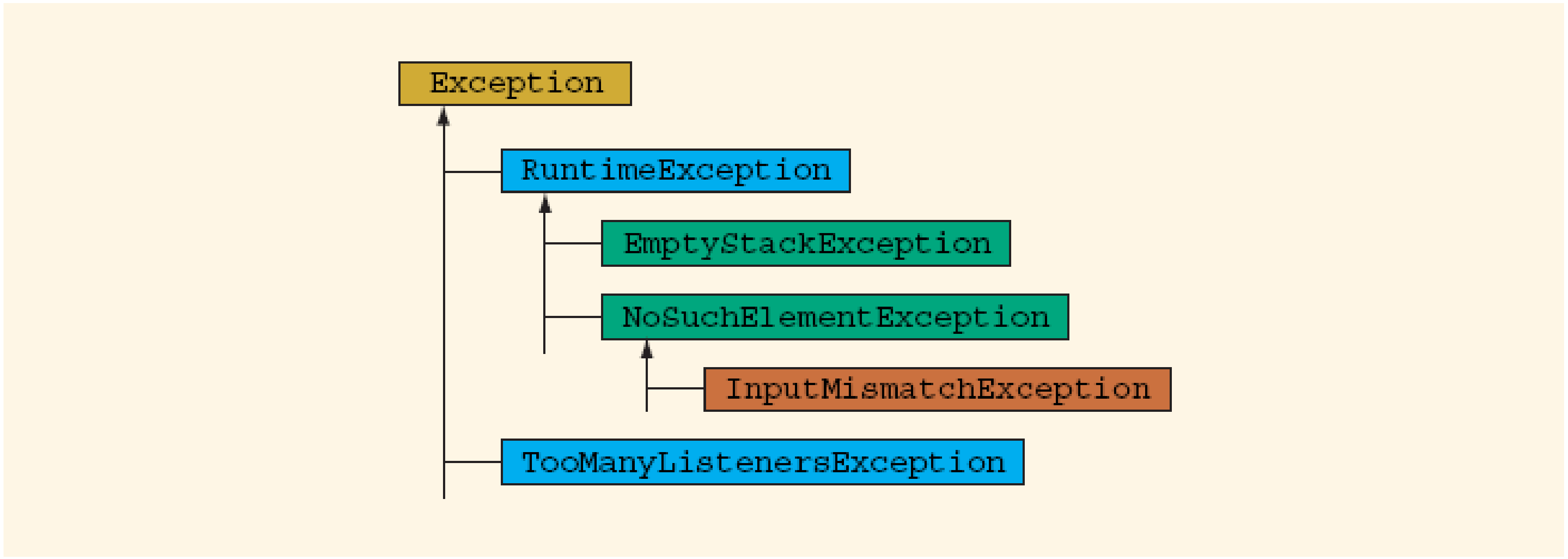
- `class Exception`
  - Subclass of `class Throwable`
  - Superclass of classes designed to handle exceptions
- There are many different types of exceptions:
  - I/O exceptions (e.g. for opening / closing files)
  - Number format exceptions
  - File not found exceptions
  - Array index out of bounds exceptions
- Exceptions categorized into separate classes and contained in various packages

Java Exception Hierarchy (continued)



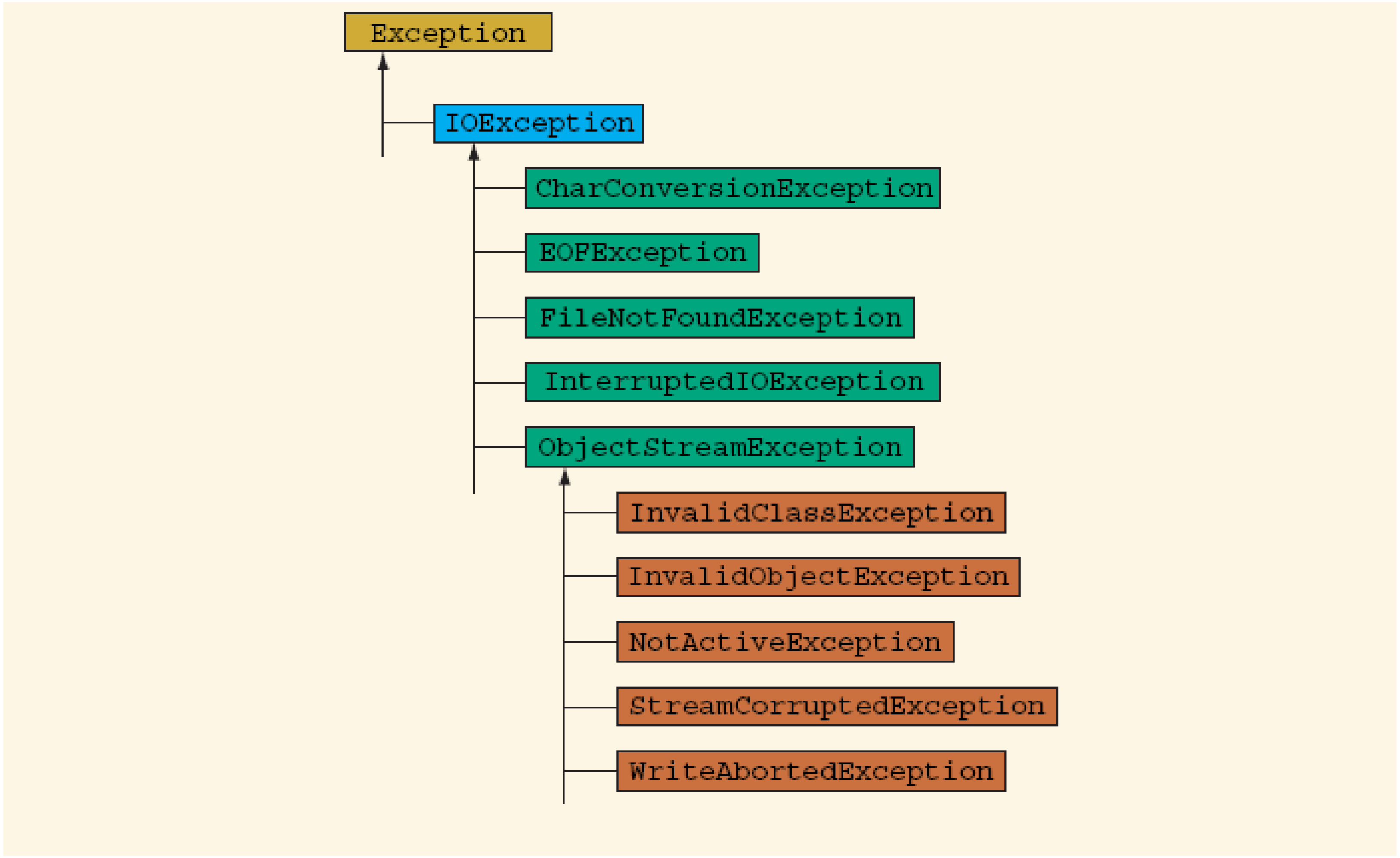
The **class** `Exception` and some of its subclasses from the **package** `java.lang`

Java Exception Hierarchy (continued)



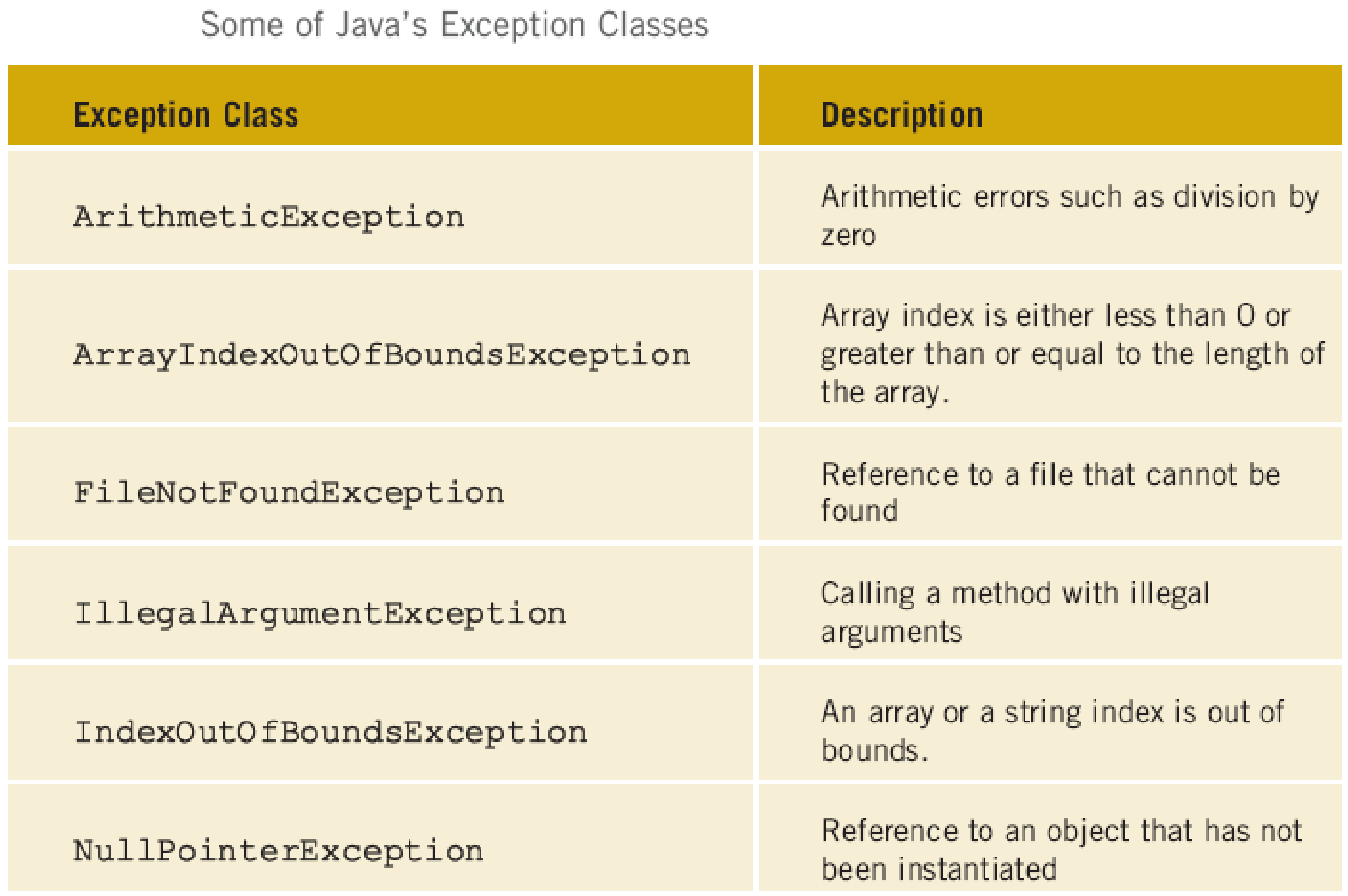
The class Exception and some of its subclasses from the package java.util. (Note that the class RuntimeException is in the package java.lang)

Java Exception Hierarchy (continued)



The class IOException and some of its subclasses from the package java.io

Some Java Exception Classes



Some of Java's Exception Classes

Exception Class	Description
`ArithmeticException`	Arithmetic errors such as division by zero
`ArrayIndexOutOfBoundsException`	Array index is either less than 0 or greater than or equal to the length of the array.
`FileNotFoundException`	Reference to a file that cannot be found
`IllegalArgumentException`	Calling a method with illegal arguments
`IndexOutOfBoundsException`	An array or a string index is out of bounds.
`NullPointerException`	Reference to an object that has not been instantiated

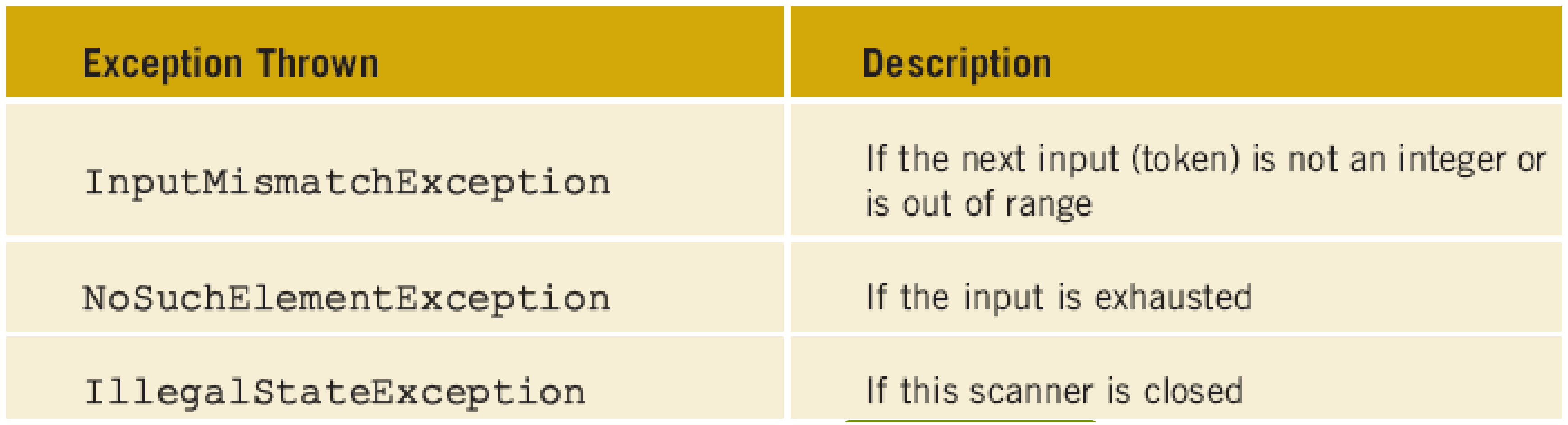
Some Java Exception Classes (contd)

Some of Java's Exception Classes (continued)

Exception Class	Description
`NumberFormatException`	Use of an illegal number format
`StringIndexOutOfBoundsException`	A string index is either less than 0 or greater than or equal to the length of the string.
`InputMismatchException`	Input (token) retrieved does not match the pattern for the expected type, or the token is out of range for the expected type.

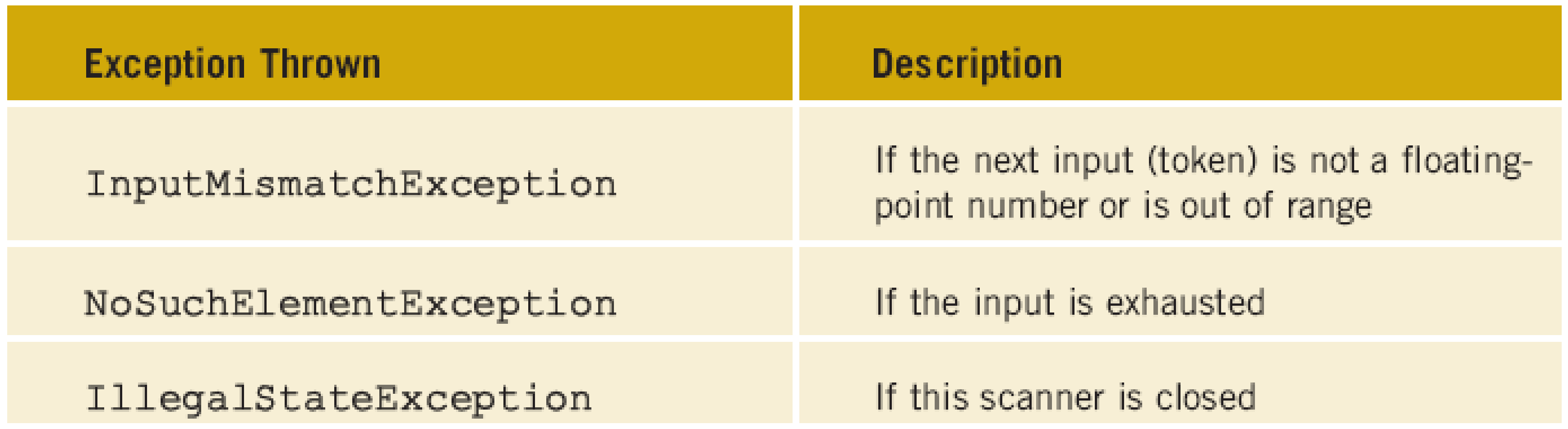
Some exceptions from the Scanner class

Exceptions Thrown by the Method `nextInt`



Exception Thrown	Description
`InputMismatchException`	If the next input (token) is not an integer or is out of range
`NoSuchElementException`	If the input is exhausted
`IllegalStateException`	If this scanner is closed

Exceptions Thrown by the Method `nextDouble`



Exception Thrown	Description
`InputMismatchException`	If the next input (token) is not a floating-point number or is out of range
`NoSuchElementException`	If the input is exhausted
`IllegalStateException`	If this scanner is closed

API for nextInt() method

Java Exception Classes (continued)

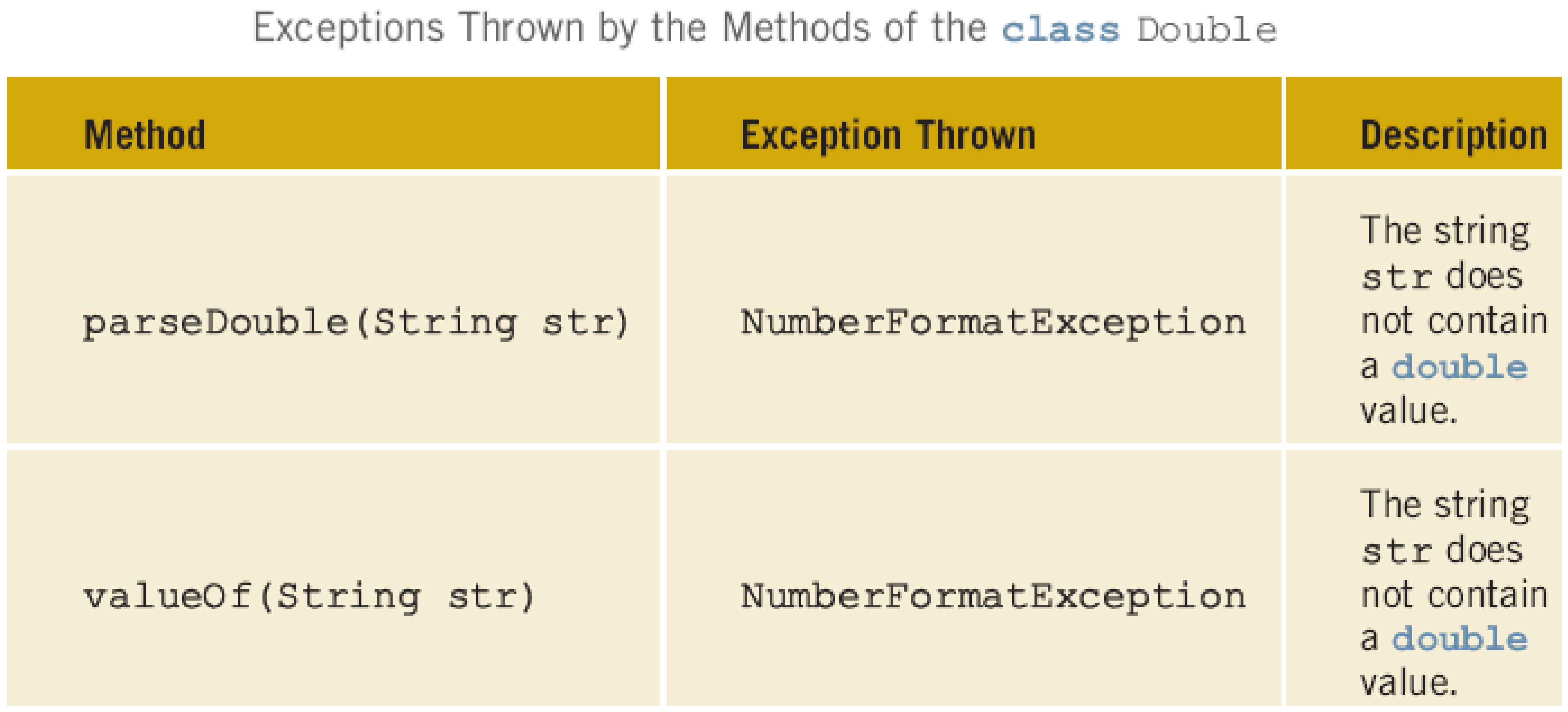
Exceptions Thrown by the Method `hasNext`

Exception Thrown	Description
`IllegalStateException`	If this scanner is closed

Exceptions Thrown by the Methods of the `class Integer`

Method	Exception Thrown	Description
`parseInt(String str)`	`NumberFormatException`	The string `str` does not contain an `int` value.
`valueOf(String str)`	`NumberFormatException`	The string `str` does not contain an `int` value.

Java Exception Classes (continued)

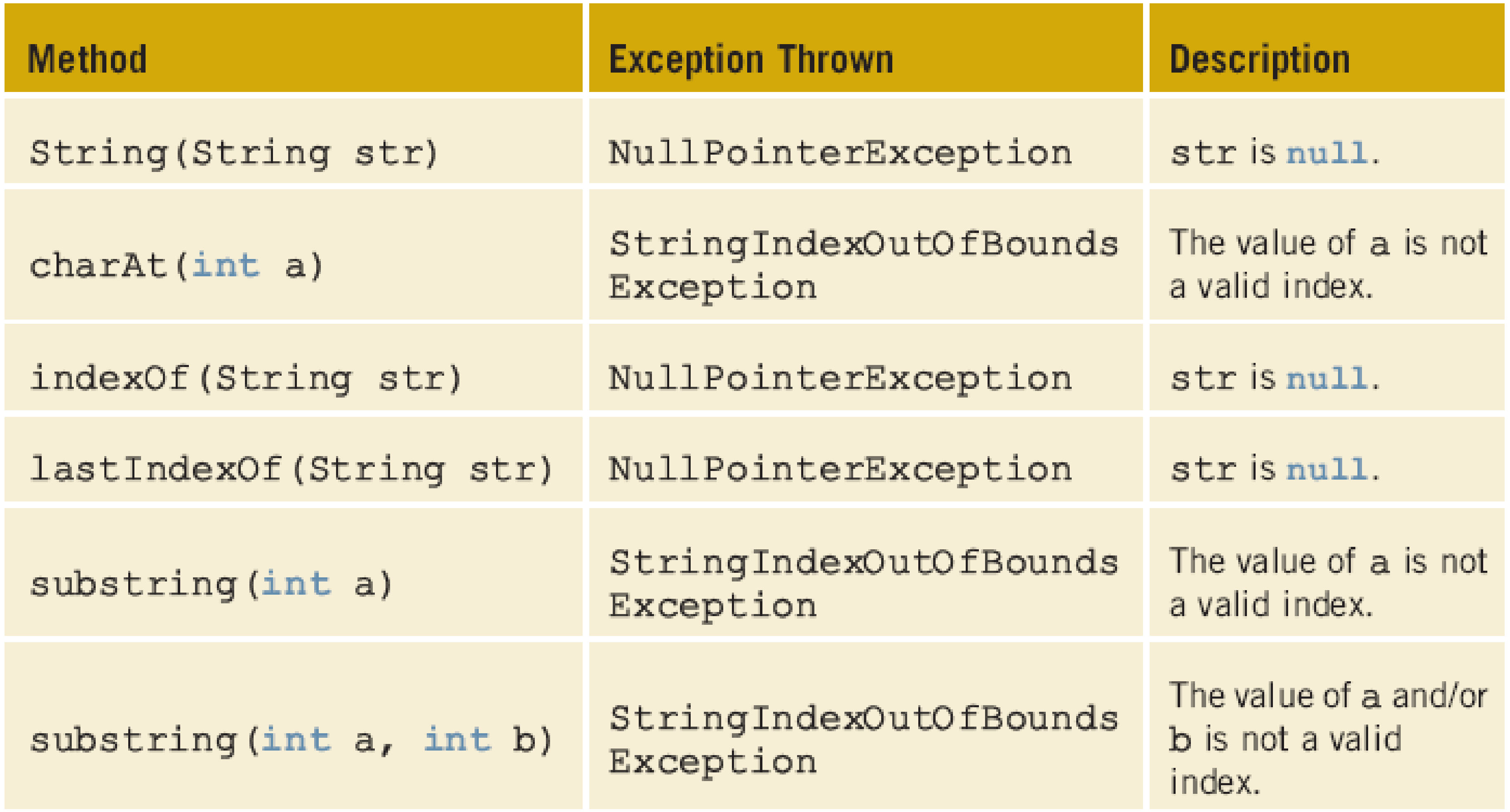


Exceptions Thrown by the Methods of the class Double

Method	Exception Thrown	Description
parseDouble(String str)	NumberFormatException	The string str does not contain a double value.
valueOf(String str)	NumberFormatException	The string str does not contain a double value.

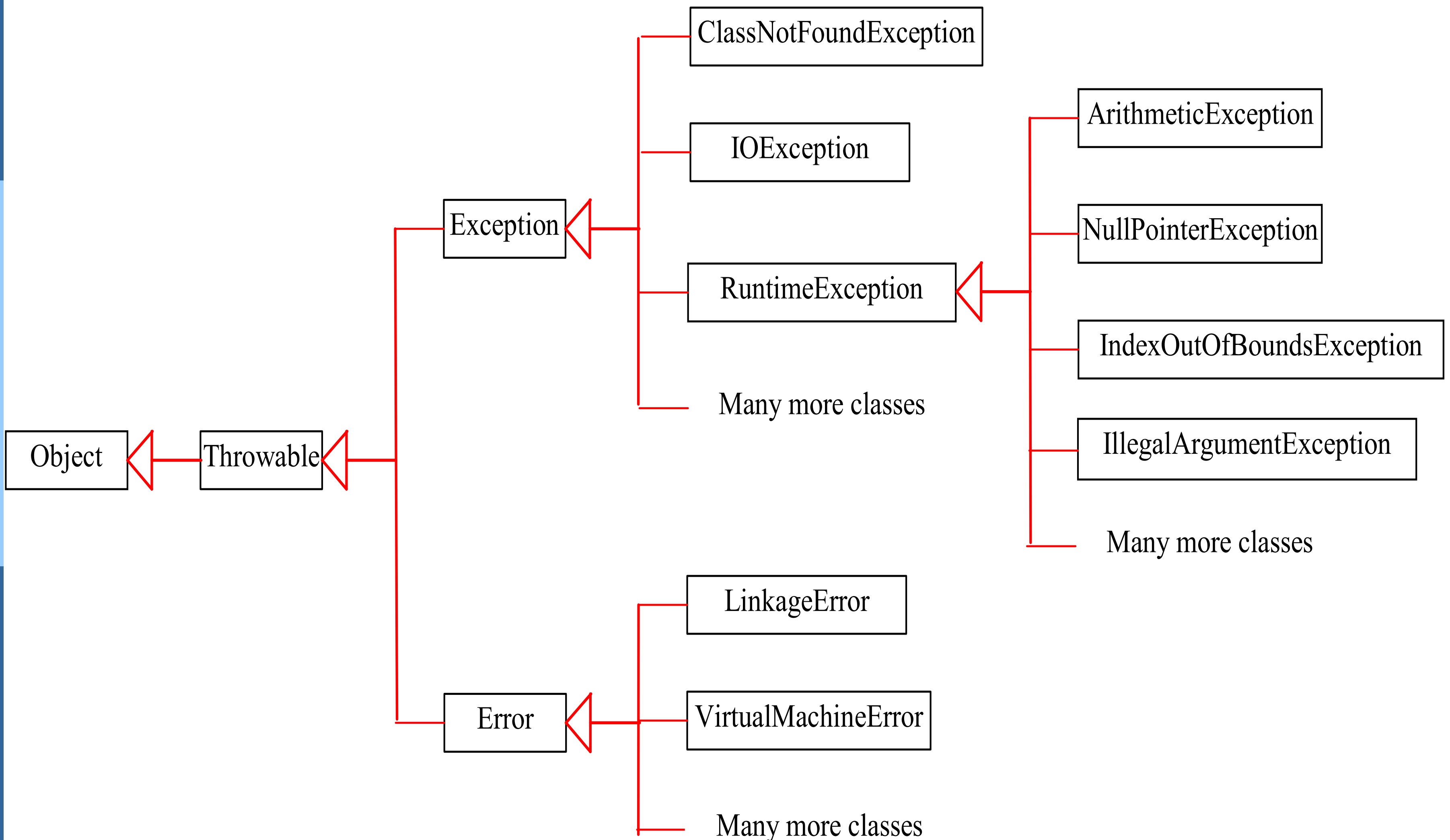
Java Exception Classes (continued)

Exceptions Thrown by the Methods of the class String

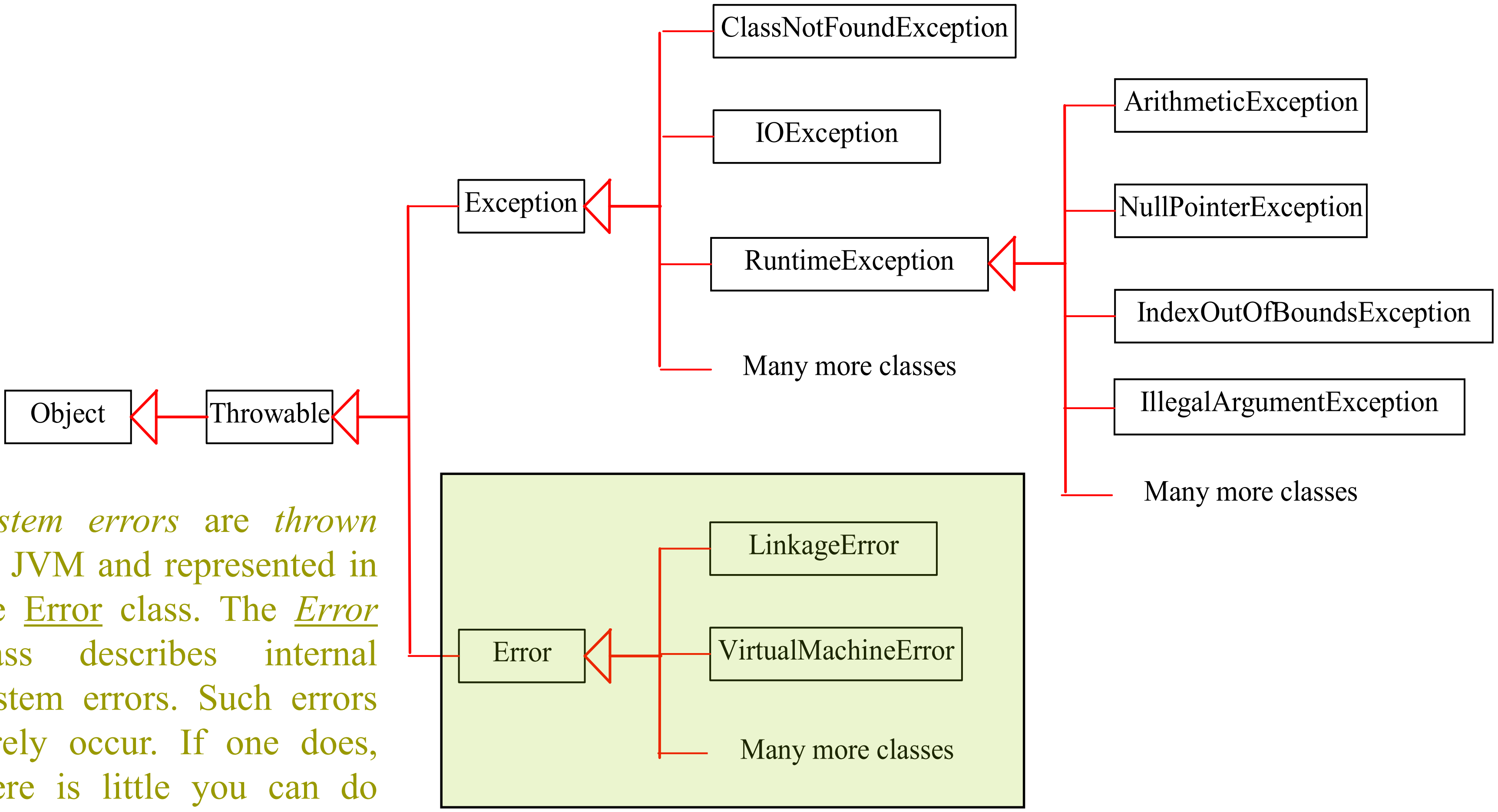


Method	Exception Thrown	Description
`String(String str)`	`NullPointerException`	str is null.
`charAt(int a)`	`StringIndexOutOfBoundsException`	The value of a is not a valid index.
`indexOf(String str)`	`NullPointerException`	str is null.
`lastIndexOf(String str)`	`NullPointerException`	str is null.
`substring(int a)`	`StringIndexOutOfBoundsException`	The value of a is not a valid index.
`substring(int a, int b)`	`StringIndexOutOfBoundsException`	The value of a and/or b is not a valid index.

Exception Types



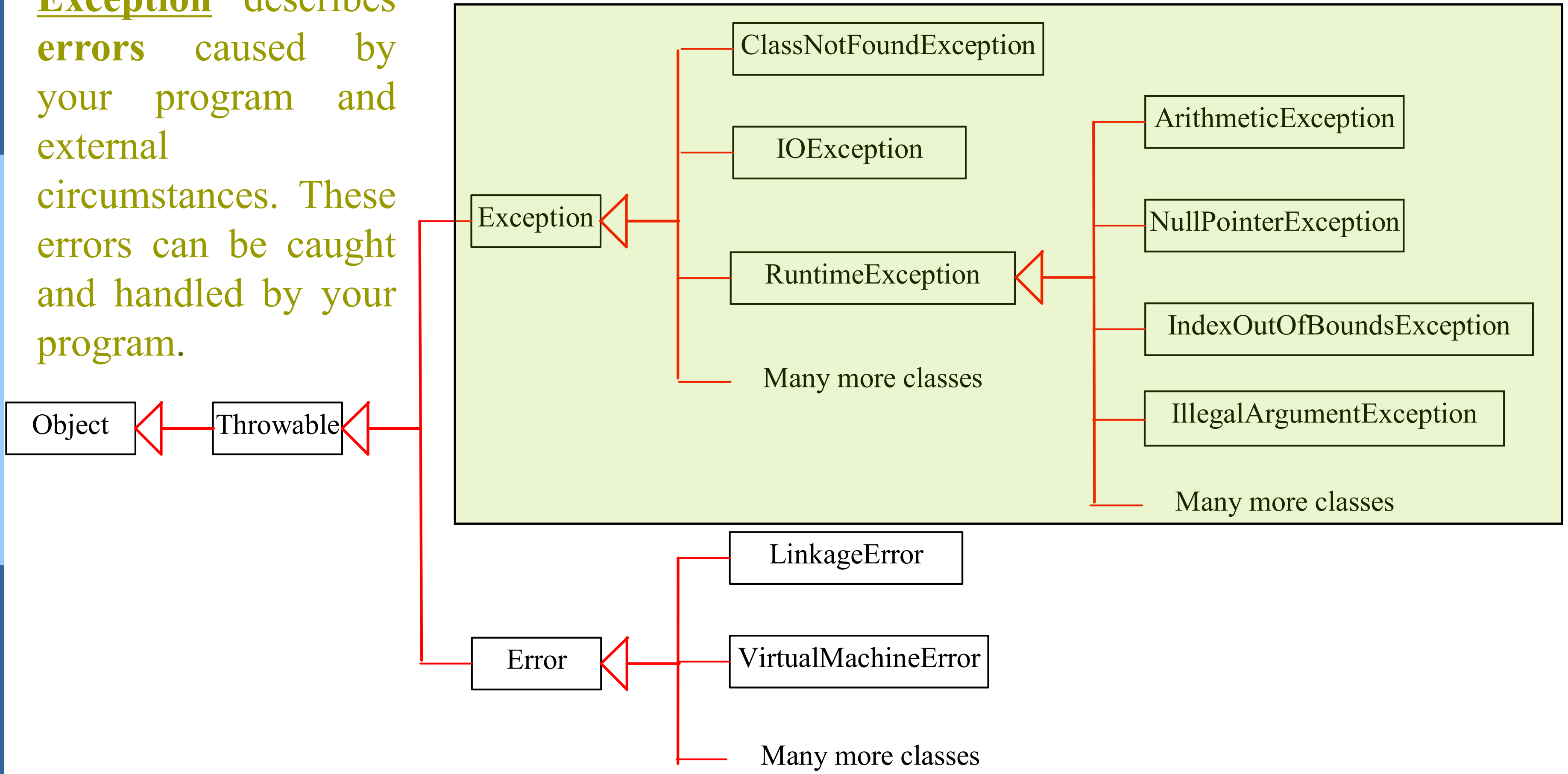
System Errors



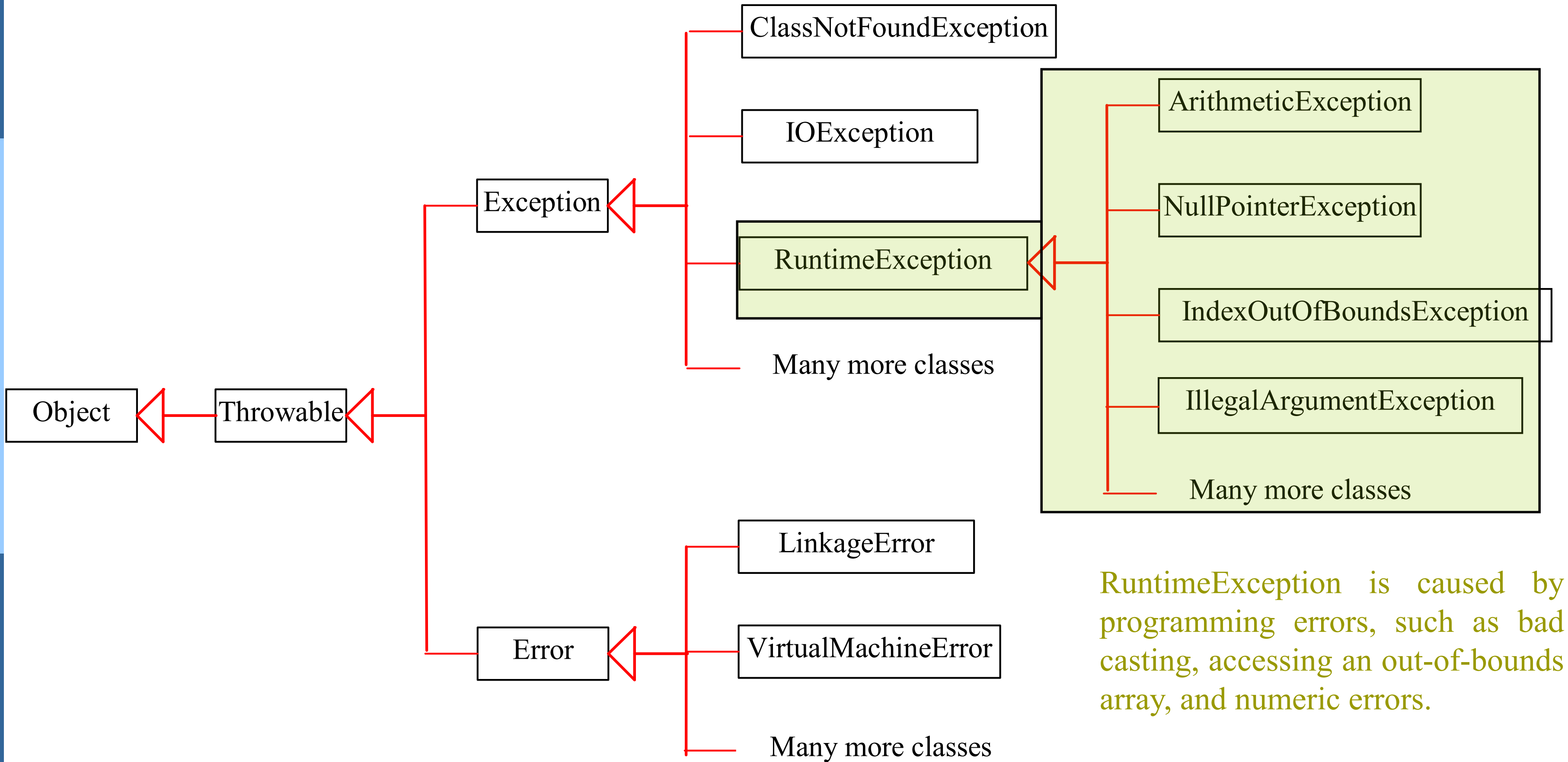
System errors are *thrown* by JVM and represented in the Error class. The *Error* class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Throwing an Exception (continued)

```
import java.util.*;
public class RethrowExceptionExmp1
{
    static Scanner console = new Scanner(System.in);
    public static void main(String[] args)
    {
        int number;
        try
        {
            number = getNumber();
            System.out.println("Line 5: number = "
                                 + number);
        }
        catch (InputMismatchException imeRef)
        {
            System.out.println("Line 7: Exception "
                                 + imeRef.toString());
        }
    }
}
```

Code is continued on next slide…

Rethrowing and Throwing an Exception (continued)

```
    public static int getNumber() throws InputMismatchException
    {         int num;
        try
        {
           System.out.print("Line 11: Enter an "
                               + "integer: ");
           num = console.nextInt();
           System.out.println();
           return num;
        }
        catch (InputMismatchException imeRef)
        {
            throw imeRef;
           // Rather than handle the exception, the
           // method RETHROWS the exception
        }
    }
}
```

The exception is rethrown to whichever line of code invoked the `getNumber()` method. The other important requirment is that the outer environment (i.e. the `main()` method in this example), is ready to handle an exception of type `InputMismatchException`

Creating Your Own Exception Classes

```
public class MyDivisionByZeroException
                                    extends Exception
{
    public MyDivisionByZeroException()
    {
        super("Cannot divide by zero");
    }

    public MyDivisionByZeroException(String
                                        strMessage)
    {
        super(strMessage);
    }
}
```