

Introduction to Programming

Part I

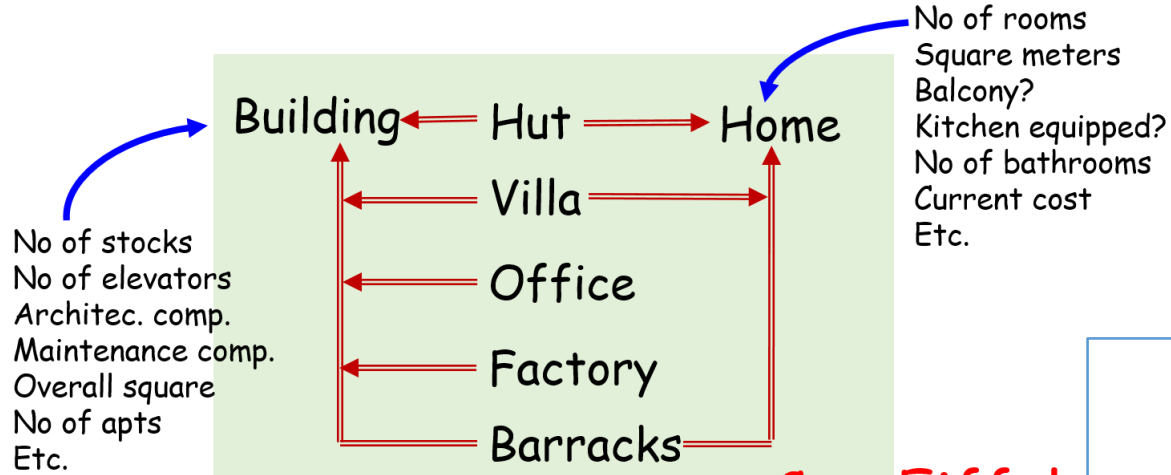
Lecture 10 Introduction to Java Exceptions

Eugene Zouev
Fall Semester 2021
Innopolis University

What We Have Learnt

- Classes and class instances
- Value types and reference types
- Encapsulation, overloading
- Inheritance: single & multiple
- Static & dynamic types
- Method overriding
- Polymorphism
- Casts & type checks
- Abstract classes & methods
- Packages

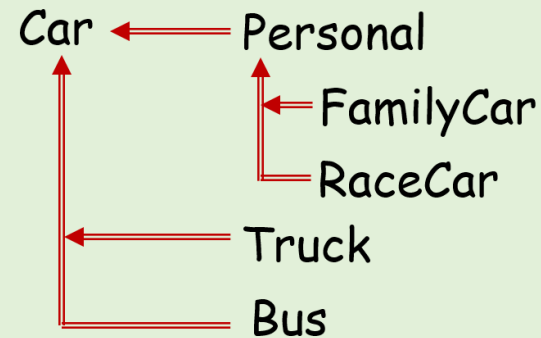
Multiple Inheritance



C++, Eiffel

"Villa" is a "Building" and is "Home" at the same time

Inheritance 3



Inheritance can be treated as "**is a**" relation:

"Personal" is a "Car"
"FamilyCar" is "Personal"
"FamilyCar" is a "Car"

Another kind of relation is **delegation**: "**has a**" relation:

"Car" has an "engine". Therefore, "Personal" and "FamilyCar" also have an "Engine" - as all other kinds of "Cars".

Static & Dynamic Types 2

Static type of `figure` is `Shape`: it is specified statically, in the program text.

```
Circle circle = new Circle();
```

```
...
```

```
Shape figure = circle;
```

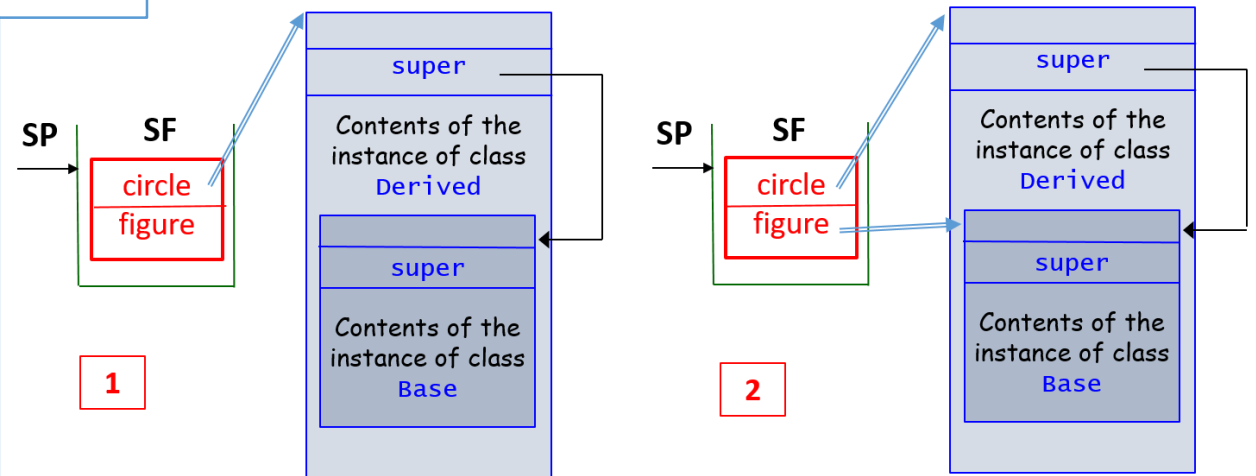
This is the conversion:
from derived type to base type

After this assignment `figure` refers to an instance of class `Circle`. It's said, that the **dynamic type** of `figure` now is `Circle`.

Static & Dynamic Types 3

```
(1) Circle circle = new Circle();
```

```
(2) Shape figure = circle;
```



The main rule of polymorphism

The interpretation of the call of a virtual method depends on the type of the object for which it is called (the dynamic type),

whereas

the interpretation of a call of a non-virtual method function depends only on the type of the reference denoting that object (the static type).

Polymorphism 7

Late binding

```
class Base
{
    public int f(int p) { return x*x; }
}

class Derived extends Base
{
    public int f(int p) { return x*x*x; }
}
```

These two methods have the same signature

This method overrides the method with the same signature from the base class

```
class SomeOtherClass
{
    public void someOtherMethod()
    {
        int result;
        Base m = new Base(); result = m.f(3);
        m = new Derived();   result = m.f(3);
    }
}
```

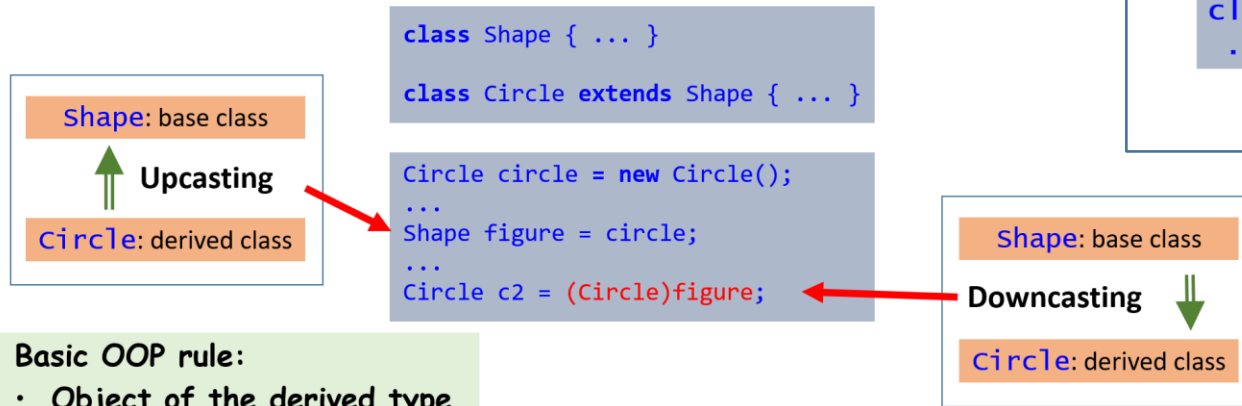
The static type of `m` is (always) `Base`

Here, the dynamic type of `m` is `Base`. The method `f` from `Base` gets called

Here, the dynamic type of `m` is `Derived`. The method `f` from `Derived` gets called!

32/35

Downcasting & Type Checks 2



Basic OOP rule:

- Object of the derived type can be converted to an object of the base type

The rule is based on the relation "is a":
Circle is a Shape hence **Circle** can be treated as **Shape**.

Upcasting: always valid

Downcasting: valid only if the instance is actually of the target type

8/27

Packages in Java 1

- Each class or group of classes can be made a member of a package:

```
package myPackage;  
class C1 { ... }  
class C2 { ... }  
...
```

This is a kind of "header" of the package called **myPackage**.

All following classes within this file are treated as members of **myPackage** package.

Full names of the classes are **myPackage.C1**, **myPackage.C2** etc. ("Fully qualified names")

Two parts of the same package

```
package myPackage;  
class C10 { ... }  
class C20 { ... }  
...
```

A package can be made up of **several files** (all residing in the same directory)

19/27

Today:

Exceptions & Exception Mechanism

Exceptions: An Introduction

- Good programmer's assumption:
“*Any code of significant length has bugs*”
- The real point, in good programming, is not to avoid bugs, i.e. errors in the executing code, but to design code that can:
 - **Detects errors** when they happen, without crashing the program or, worse, the system (*fault awareness*)
 - **Recovers from an error**, giving the user a choice to continue using the program and to save the greatest possible amount of work already done (*fault recovery*)
 - **Keeps on running consistently** notwithstanding an error, still performing activities that are not affected by the error (*fault tolerance*)

Exceptions: Error Handling

- All these issues, from a programmer's perspective, are collectively known as **error handling**
 - Of course the best would be to have any portion of a program to be fault tolerant
 - But usually we have to be happy if only the critical parts of the code are fault tolerant (or, at least, able to recover from an error), while the others are only fault aware
- During the decades, from the early days of programming onwards, many different techniques have been devised to cope with errors

Error Handling: Classic Approach 1

```
double squareRoot(double v)
{
    Calculations
    return someResult
}
```

What to do if $v < 0$??

```
...
if ( aValue >= 0 )
    res = squareRoot(aValue)
else
    ... // ???
```

Still the problem: What to do if $aValue < 0$??

```
double squareRoot(double v)
{
    if ( v < 0 ) return -1;
    Calculations
    return someResult;
}
```

Approach with error codes

```
res = squareRoot(aValue);
if ( res == -1 )
    Do something
...
```

Error Handling: Classic Approach 2

- A common way of managing error handling, in pre-OO era languages, as C or Pascal, was to return an error code to the caller of a function, either using the return value of the function itself or an additional output parameter.
- This was (and still is) an **extremely error-prone technique**, because it relied on the user of a method (“the caller”) to be willing to test the error code to see if the function had performed flawlessly
- Programmers often forgot, or neglected, to test error codes, thus going ahead blindly, not really knowing whether their code was or not in an consistent state

The C language and (surprisingly) Go still use this technique

Error Handling: Classic Approach 3

- Another issue with this kind of error management was its **excess of locality**
 - sometimes the caller doesn't have enough information to recover from an error issued from a function
- Therefore the solution usually were:
 - passing the error code further up the calling chain - where the context hopefully allowed meaningful recovery
 - calling global error handling routines through **goto**-like instructions (ugh!)
 - using global variables ☹️

```
int f() {  
    ...  
    res = squareRoot(aValue);  
    if ( res == -1 )  
        // The f function might not  
        // know what to do with the  
        // situation...  
        return -1;  
        // Therefore we have to pass  
        // the error code further  
}  
...
```

Error Handling: Classic Approach 4

- All this led to programs where normal code and error management code were highly coupled and intermixed
 - Therefore, these programs were also rather obscure, i.e. extremely difficult to read, unless very well documented
- => This was one of the problems which led to the definition of the concept of exception

The Concept of Exception

The Concept of Exception

- In a very general sense, an exception is any event that does not belong to the normal flow of control of a program
 - if an event has to be considered an exception is not always clear cut, but often depends on the specific application and level of abstraction
- An exception could be **an error**
 - e.g. running out of memory, a wrong argument value, etc.
- But **this is not always the case**, an exception could also be, e.g.:
 - an **unusual result** from a math computation routine
 - an **unexpected (but not wrong) request** to an operating system
 - the **detection of an input from the “external world”** (say an interrupt request, a login request, a power shortage, etc.)

The Concept of Exception

Trivial case

```
for (int i=0; i<=N; i++) {  
    ...  
    if ( unexpected situation )  
        break;  
    ...  
}
```

Another case

```
int f() {  
    for (int i=0; i<=N; i++) {  
        ...  
        if ( unexpected situation )  
            return -1;  
        ...  
    }  
    return normalResult;  
}
```

Non-trivial case

```
void g()  
{  
    ...  
    res = f();  
    Process the valid result  
    ...  
    Process the unexpected result  
}  
int f() {  
    for (int i=0; i<=N; i++) {  
        for ( int j=0; j<=M; j++ {  
            ...  
            if ( unexpected situation )  
                ?????;  
            ...  
        }  
    }  
    return normalResult;  
}
```


Exceptions & OOP 1

- Exceptions are not a concept intrinsic to the OO philosophy, but every modern OO language implements this concept in some way.
- In the context of OO languages, what is known as **exception handling** becomes a powerful mechanism to implement more **robust, reliable, readable and reusable code**.
- In this context **exceptions are objects**, i.e. instances of some class, and they are used to pass information about unusual events to and from the different parts of the program that are designed to cope with them.

Each OO language supports exception mechanism

Ada:
Exceptions are not objects

Java, C#:
Exceptions are objects of a class type

C++:
Exceptions are objects of any type

Exceptions & OOP 2

Three aspects of exceptions in OO Languages:

- An **event** that breaks the “normal” flow of program control.
 - An event can cause either by run-time or by the program itself (i.e., by a special language construct).
- **Transfer** the control to some other program point.
 - The way of transferring control is defined by the language semantics.
- An **object** that is passed together with transferring the control.
 - The object passed is **an instance of some class** declared in the program (or in the standard library).
- It is useful to compare this with the usual function call/return.

Exception Mechanism 1

The first aspect: breaking the "normal" flow of program control.

`throw ref-to-object`

An event can cause either by run-time support or by the program itself.

In both cases the function breaks its execution

```
void f() {  
    ...  
    int d = 0;  
    int x = (a+b*c)/d;  
    ...  
}
```

System-defined exception occurs. The reason is "zero division".
No need to explicitly specify `throw`.

```
void roots(float a, float b, float c)  
{  
    if ( a == 0.0 ) throw new NotSqrEqu();  
    float d = b*b-4*a*c;  
    if ( d < 0 ) throw new NoRatRoots();  
    ...  
}
```

User-defined exceptions occur. The reasons belong to the program logic.

Exception Mechanism 2

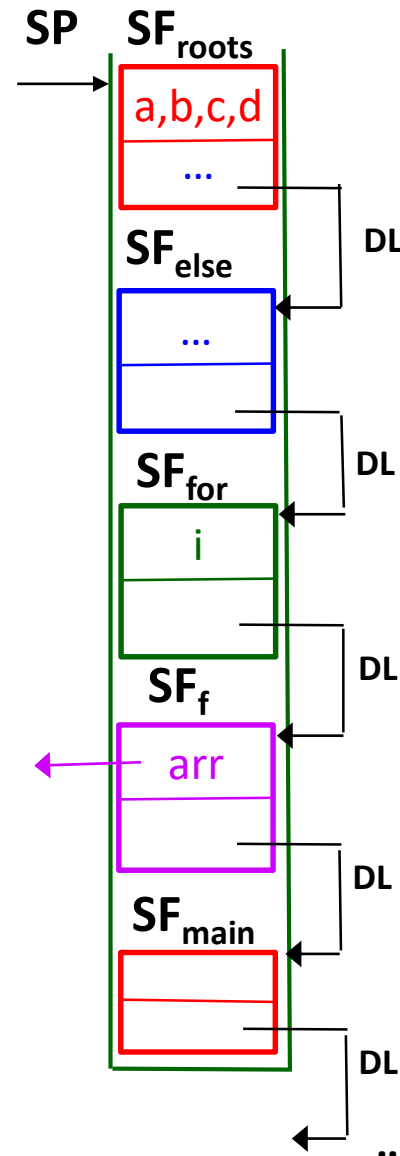
The second aspect: transfer the control to some other program point.

The main idea & the main rule about transferring the control is:

The control is transferred sequentially to dynamically enclosing scopes: from the current scope up to the outermost scope ...until the transfer finds a scope specially specified for "catching" the exception

Exception Mechanism 3: Example

```
void main()
{
    ... f(); ...
}
void f()
{
    double[] arr = new double[20];
    ...
    for (int i=0; i<20; i++)
    {
        if ( condition )
            // Some other calculations
        else {
            roots(i,arr[i],arr[i+1]);
        }
    }
}
void roots(float a, float b, float c)
{
    if ( a == 0.0 ) throw new NotQuadEqu();
    float d = b*b-4*a*c;
    if ( d < 0 ) throw new NoRatRoots();
    ...
}
```



Stack unwinding

The normal control flow gets interrupted, and all stackframes are removed from the stack.

Exception propagation

All dynamically called functions and blocks are terminated.

When the unwinding process stops???

Exception Mechanism 4

Try block: where exceptions are caught

This is simply a **marker** of the try clock

From a previous slide:

The control is transferred sequentially to **dynamically enclosing scopes**: from the current scope up to the outermost scope


...until the transfer finds a scope specially specified for **"catching" the exception**

This is a usual block:

A block where an exception can be thrown

This is a **catch-handler**: specifies the algorithm for processing exceptions. Looks similar to the function declaration

There can be **more than one** catch handler in the try block

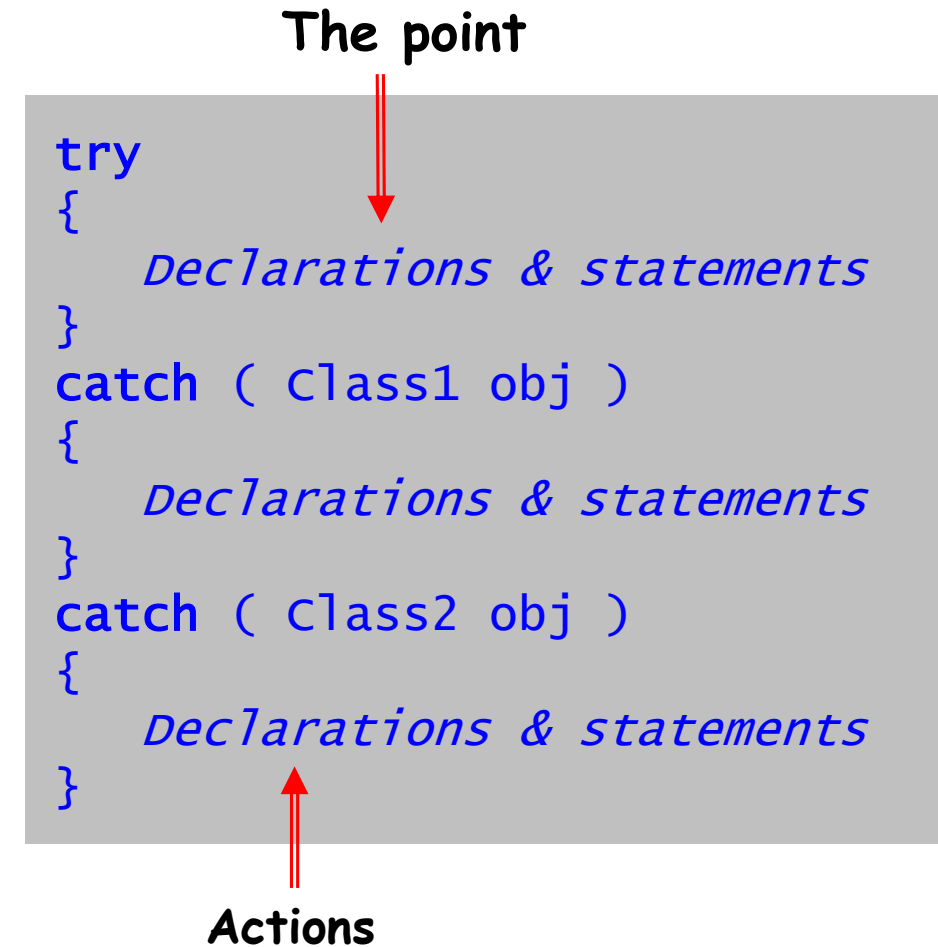


```
try
{
    Declarations & statements
}
catch ( Class1 obj )
{
    Declarations & statements
}
catch ( Class2 obj )
{
    Declarations & statements
}
```

Exception Mechanism 5

Semantics of try block

- The try block determines the **point** in the program code where exceptions get caught.
- The try block specifies **actions** to be done after the exception is caught ("reaction").
- The selection of actions that are to be performed after catching an exception is based on **the class specified in the handlers**.
- Syntactically, try block is a **statement**. Thus, it can be used in any place where usual statement can be used.

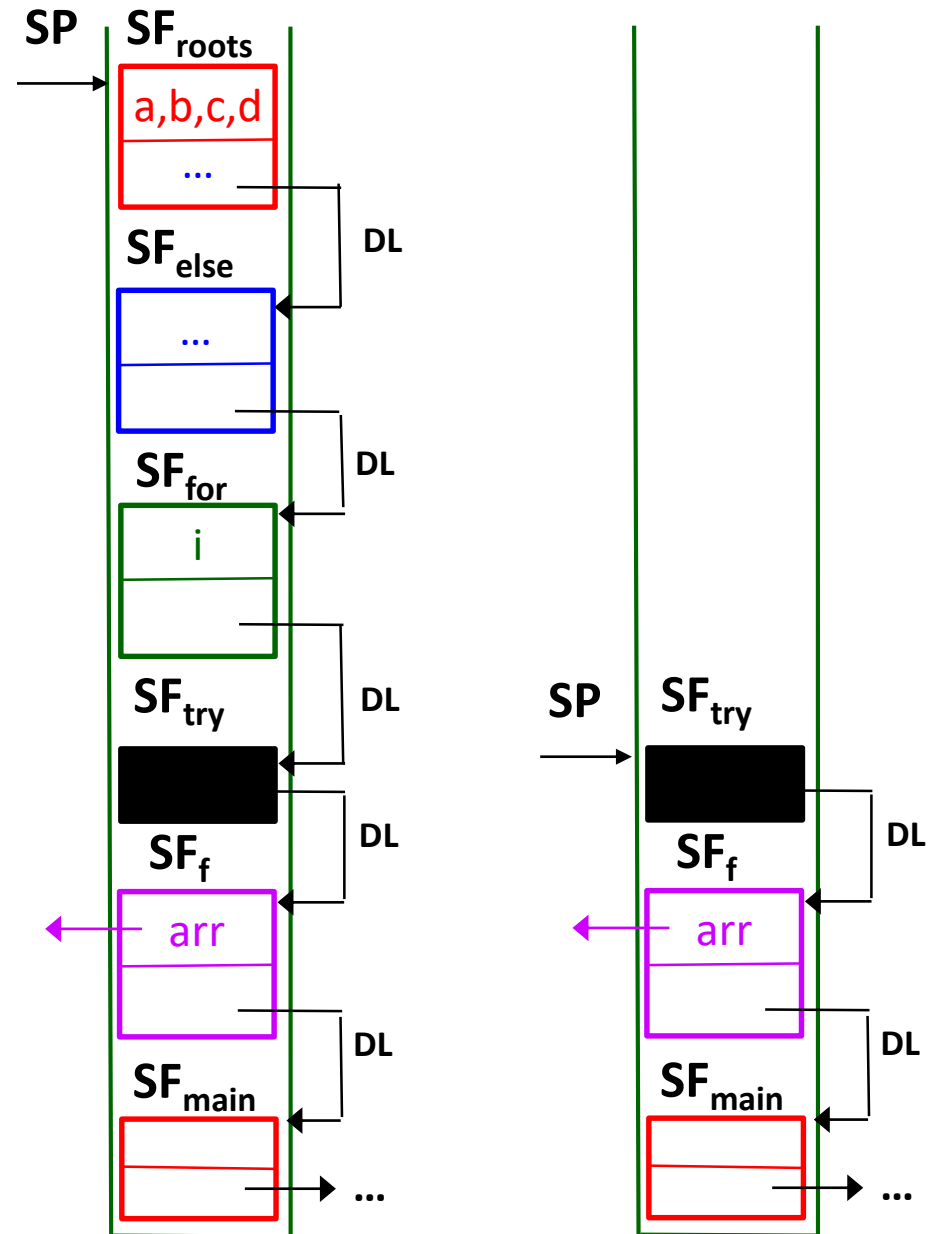


Exception Mechanism 6: Example

```
void main() { ... f(); ... }

void f()
{
    double[] arr = new double[20];
    ...
    try {
        for (int i=0; i<20; i++)
        {
            if ( condition )
                // some other calculations
            else {
                roots(i,arr[i],arr[i+1]);
            }
        }
    }
    catch ( NotSqrEqu o ) { ... }
    catch ( NoRatRoots o ) { ... }
    ...
}

void roots(float a, float b, float c)
{
    if ( a == 0.0 ) throw new NotQuadEqu();
    float d = b*b-4*a*c;
    if ( d < 0 ) throw new NoRatRoots();
    ...
}
```



Exception Mechanism 7

Selection of a suitable catch-handler

```
try
{
    ...
}
catch ( Class1 obj )
{
    ...
}
```

The main rules:

- The **catch handler** processes the exception thrown with the instance of type **Class1** or of **any class derived from Class1**.
- The value of the exception thrown is passed to the handler exactly as a method parameter (i.e., by reference).
- After exiting from the handler the program execution continues normally.

Remember dynamic types principle & upcasting!

Exception Mechanism 9

Selection of a suitable catch handler

The first catch handler catches all exceptions of type `Animal` and of all classes derived from it - including `Lion`!

If there is no handler suitable for the exception thrown with the object of a particular type then the process is continued.

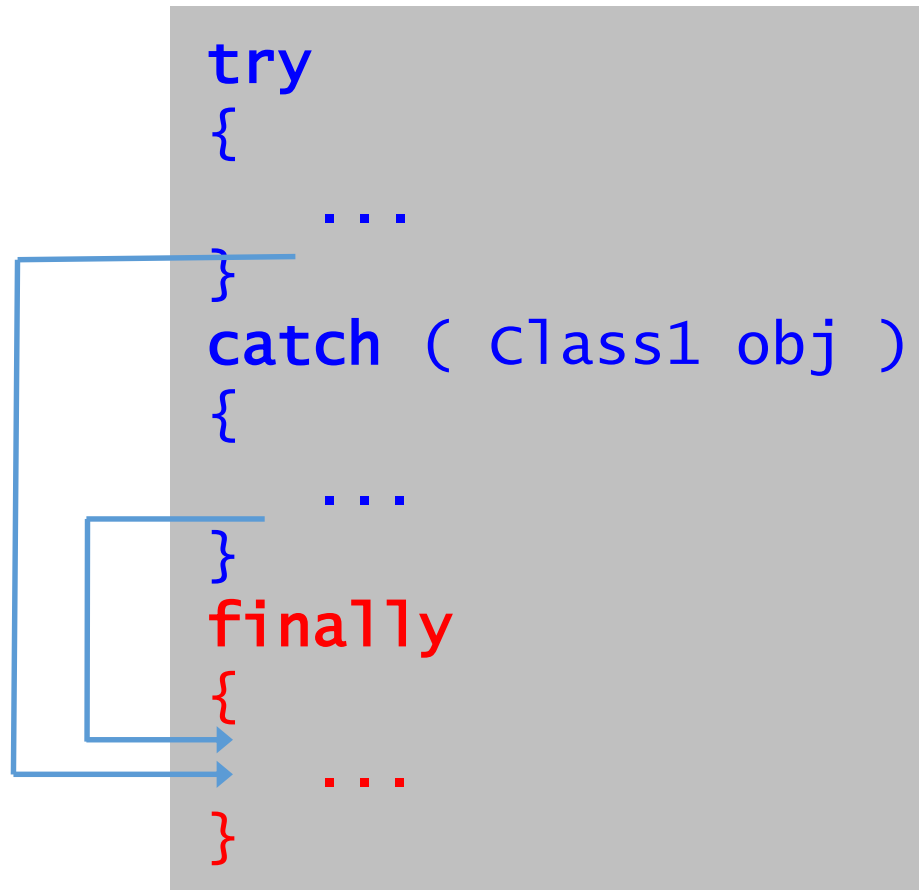
```
class Animal { ... }
class Lion extends Animal { ... }
...
try
{
    throw new Lion();
    ...
    throw new Animal();
    ...
    throw new Car();
}
catch ( Animal obj )
{
    ...
}
catch ( Lion obj )
{
    ...
}
```

Directly in this try block
OR in a dynamically
enclosed block/function

Therefore, the second
handler is **never activated**!

Exception Mechanism 10

Finally clause



```
try
{
    ...
}
catch ( Class1 obj )
{
    ...
}
finally
{
    ...
}
```

The diagram illustrates the execution flow of a try-catch-finally block. Blue arrows show the sequence: first, the code inside the try block is executed; then, if an exception is caught, the code inside the catch block is executed; finally, the code inside the finally block is executed. The finally block is highlighted in red in the original image.

C++: no **finally**

The rules:

The code inside the finally block is always executed:

- After execution of any catch handler.
- In case there is no exception thrown from within the try block.
- When stack unwinding process is performed, and no suitable exception were found.

Finally block is used for actions that should be performed in any case.

Exception Specification

In addition to the exception mechanism itself (**throw-try-catch-finally**), it can be useful to specify the set of exceptions **that might be thrown by a function** - **as part of the function declaration**.

It doesn't return any value...

The function can be called by its name f...

...And it can throw exceptions of type T1 or T2!

```
void f(int x) throws T1, T2
{
    ...
}
```

It accepts one parameter of integer type...

Throws clause: it's a part of function prototype (header)

Exception specification allows the compiler to ensure that code for handling exceptions has been included.

The similar feature was **removed** from the C++ standard 😊

Java Exception Classes 1

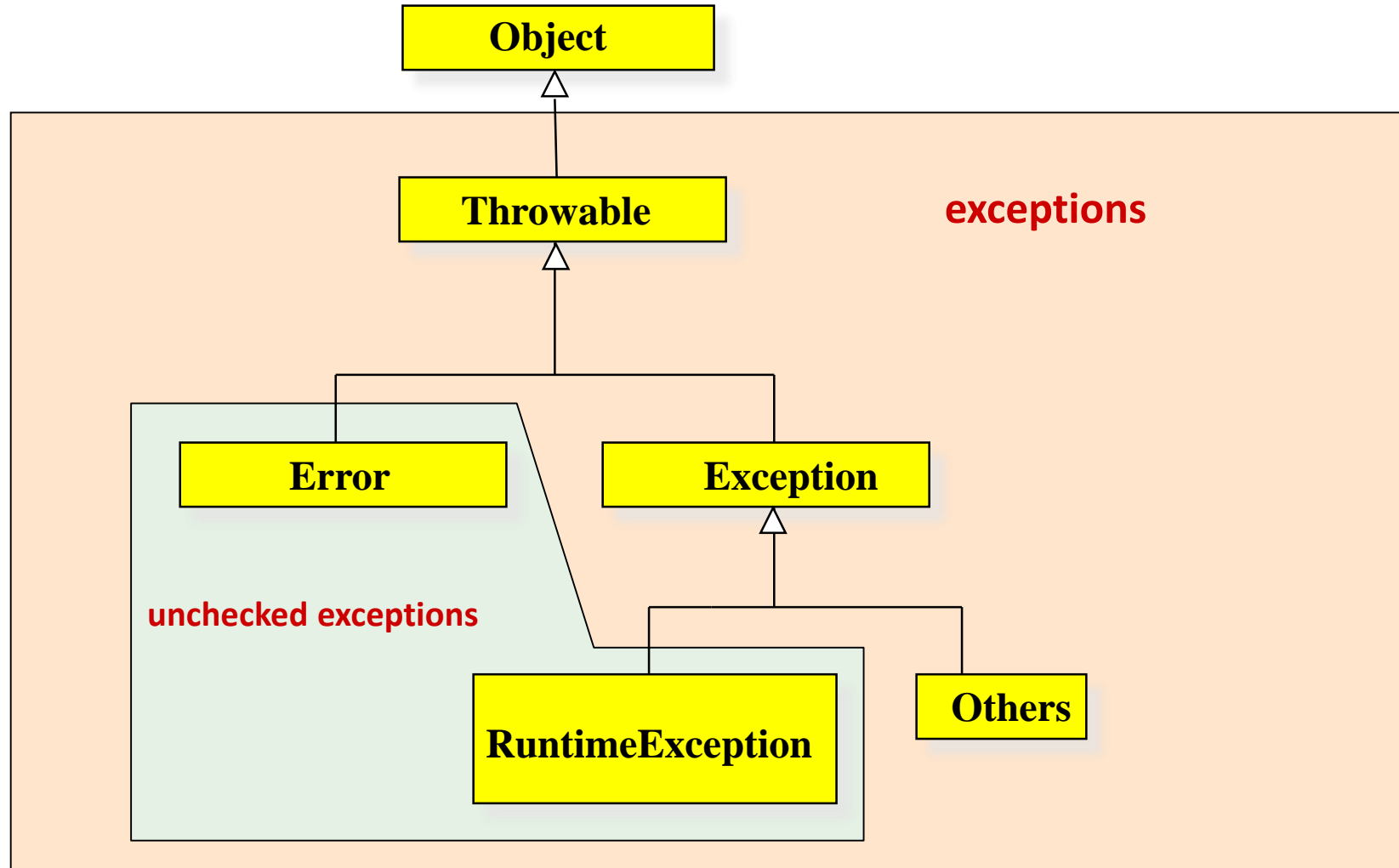
- Every class used in the exception mechanism must be derived from a special class declared in the Java standard library.
- There are two kinds of "exception classes" in Java: **checked** exception classes and **unchecked** exception classes.
- Unchecked exception classes should be derived from `java.lang.RuntimeException` or `java.lang.Error`.
- Checked exception classes should be derived from the `java.lang.Throwable` class.

This means that objects passed via exceptions should be class instances

C++: exception objects can be of any type: e.g., `int`, `enum` etc.

Java Exception Classes 2

The hierarchy of exception classes



Java Unchecked Exceptions 1

- The use of unchecked exceptions usage impose **no extra constraint** in writing the code.
- A programmer can define her/his own unchecked exception classes by subclassing **Error** (not advisable) or **RuntimeException**
 - **Error** is intended as a base class for all the exceptions thrown by the system (operating system, virtual machine, etc.) so it is **not advisable to use it as a superclass for user-defined exceptions**
 - **RuntimeException** is intended as a base class for exceptions that would impose an unbearable burden on the programmer, if they were implemented as checked exceptions, especially in the java API packages

Java Unchecked Exceptions 2

The library-defined exception classes

Runtime exceptions are not required to be caught; e.g.,

✖ `ArrayIndexOutOfBoundsException`

✖ `NullPointerException`

Java Checked Exceptions

- Java **checked exceptions** usage imposes stricter constraints on how to write the code.
- Any function whose execution may give rise to a checked exception has to declare it in its **throws clause**.
- Any method that causes a checked exception to occur must
 - either catch the exception,
 - or specify the type of the exception (or a superclass of it) in the throws clause of its declaration.
- Any attempt to violate the contract provided by the exceptions specification will **provoke an error during the compilation**
- A programmer can **define her/his own checked exception classes by subclassing Throwable** (not advisable) or (better) **Exception**.