Principles of programming languages

Lecture 7

http://few.vu.nl/~nsilvis/PPL/2007

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

vrije Universiteit *amsterdam*

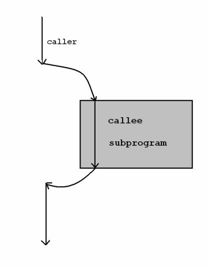
What is abstraction?

Abstraction in programming

- Abstraction=a representation of an entity that includes only significant attributes.
- A weapon against the program complexity
- The entity is thought in terms of its purpose rather than its implementation
- Control/process abstraction – performs a well-defined operation = subprograms
- Data abstraction – represents information = objects

Control abstraction

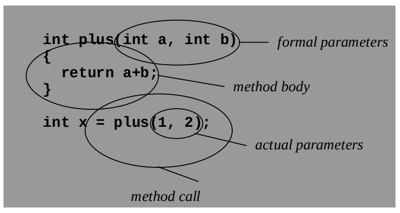
- Each subprogram has one entry point
- The calling program (caller) is suspended during execution of the called subprogram (callee)
- Control returns to the caller when subprogram is terminated



Outline

- Parameters
- Error handling

Subprograms



```
int plus(int a, int b)
{
  return a+b;
}

int x = plus(1, 2);
```

Let's talk about parameters.

Questions:

➡ How are parameters matched?

How are parameters passed from the caller to the callee?

Parameter correspondence

- How does the language match up actual and formal parameters?

```
int plus (int a, int b)
{
return a+b;
}
int x = plus (1,2) ;
```

?

Positional Parameters

- Most common case
  - Correspondence determined by positions
  - *n*th formal parameter matched with *n*th actual

```
int plus (int a, int b)
{
return a+b;
}
int x = plus (1,2) ;
```

Keyword Parameters

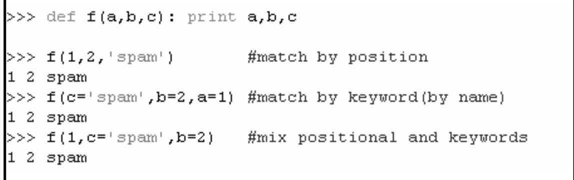
- Correspondence can be determined by matching parameter names
- Ada:

  `DIVIDE(DIVIDEND => X, DIVISOR => Y);`
- Matches actual parameter `X` to formal parameter `DIVIDEND`, and `Y` to `DIVISOR`
- Parameter order is irrelevant here

Mixed Keyword And Positional

- Most languages that support keyword parameters allow both: Ada, Fortran, Dylan, Python
- The first parameters in a list can be positional, and the remainder can be keyword parameters

Example: Python



```
>>> def f(a,b,c): print a,b,c

>>> f(1,2,'spam')        #match by position
1 2 spam
>>> f(c='spam',b=2,a=1)  #match by keyword(by name)
1 2 spam
>>> f(1,c='spam',b=2)    #mix positional and keywords
1 2 spam
```

Optional parameters in C++

```
int f(int a=1, int b=2, int c=3) { body }
```

```
int f()  -> f(1,2,3)
int f(int a) -> f(a,2,3);
int f(int a, int b) -> f(a,b,3)
int f(int a, int b, int c) {body}
```

Unlimited Parameter Lists

- C, C++, and scripting languages like JavaScript, Python, and Perl
- A hole in static type systems, since the types of the excess parameters cannot be checked at compile time

```
int printf(char *format, ...) { body }
```

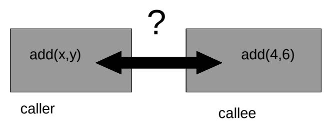
any nr. of parameters

Let's talk about parameters.

Questions:

How are parameters matched?

How are parameters passed between the caller and the callee?



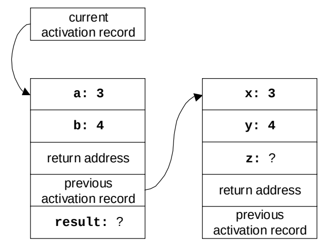
By Value

For by-value parameter passing, the formal parameter is just like a local variable in the activation record of the called method, with one important difference: it is initialized using the value of the corresponding actual parameter, **before** the called method begins executing.

- Simplest method, widely used
- The only method in real Java
- Changes to a formal do not affect the actual
- Info from caller to callee

```
int plus(int a, int b) {
  a += b;
  return a;
}
void f() {
  int x = 3;
  int y = 4;
  int z = plus(x, y);
}
```

When **`plus`** is starting


current activation record
a: 3
b: 4
return address
previous activation record
result: ?
x: 3
y: 4
z: ?
return address
previous activation record

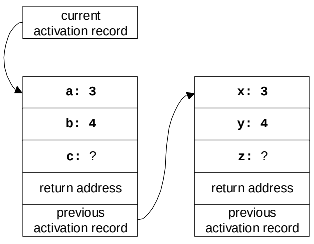
By Result

For by-result parameter passing, the formal parameter is just like a local variable in the activation record of the called method—it is uninitialized. **After** the called method finished executing, the final value of the formal parameter is assigned to the corresponding actual parameter.

- Also called *copy-out*
- Info only from callee to caller
- Introduced in Algol 68; sometimes used for Ada

Warning: the following examples might be from hypothetical languages.

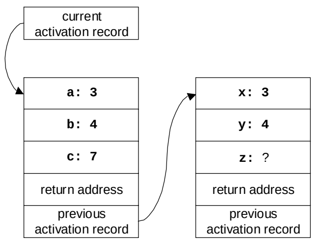
```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



current activation record
a: 3
b: 4
c: ?
return address
previous activation record
x: 3
y: 4
z: ?
return address
previous activation record

When **plus** is starting

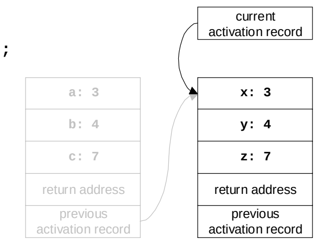
```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



current activation record
a: 3
b: 4
c: 7
return address
previous activation record
x: 3
y: 4
z: ?
return address
previous activation record

When **plus** is ready to return

```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



current activation record
a: 3
b: 4
c: 7
return address
previous activation record
x: 3
y: 4
z: 7
return address
previous activation record

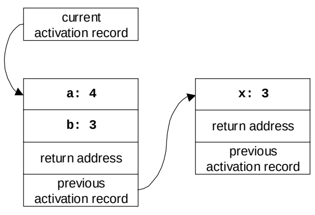
When **plus** has returned

By Value-Result

For passing parameters by value-result, the formal parameter is just like a local variable in the activation record of the called method. It is initialized using the value of the corresponding actual parameter, **before** the called method begins executing. Then, **after** the called method finishes executing, the final value of the formal parameter is assigned to the actual parameter.

- Also called *copy-in/copy-out*
- Info in both senses

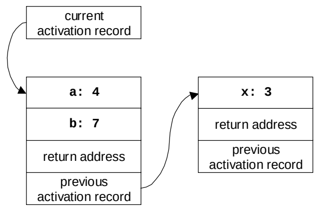
```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



current activation record
a: 4
b: 3
return address
previous activation record
x: 3
return address
previous activation record

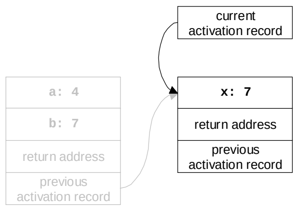
When **plus** is starting

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```

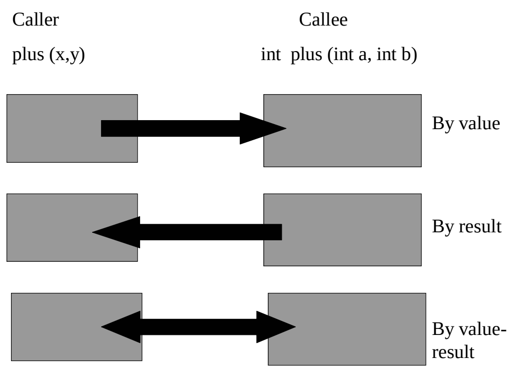


When **`plus`** is ready to return

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **`plus`** has returned



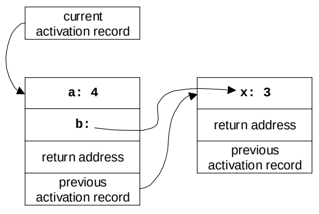
What is the disadvantage of these 3 parameter passing methods?

By Reference

For passing parameters by reference, the address of the actual parameter is computed before the called method executes. Inside the called method, this address is used as the address of the corresponding formal parameter. In effect, the formal parameter is an **alias** for the actual parameter—another name for the same memory location.

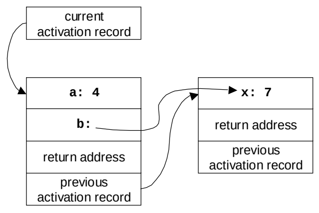
- One of the earliest methods: Fortran
- Most efficient for large objects (arrays)
- The address of the actual is passed

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **`plus`** is starting

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```


current activation record
a: 4
b:
return address
previous activation record
x: 7
return address
previous activation record

When **`plus`** has made the assignment

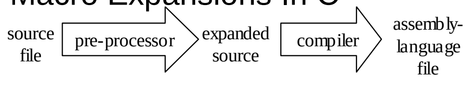
What is the difference between by ref and by value-result?

By-ref Implementations

- Early Fortran had only by-ref. Even a famous defect: F(2) is possible
- In Java all objects, including arrays are passed by reference
- C uses pass by value, and pass by ref is implemented by using pointers:

```
void plus(int a, int *b)
{*b += a; }
void f() {
  int x = 3;
  plus(4, &x);
}
```

Macro Expansions In C



- An extra step in the classical sequence
- Macro expansion before compilation

source file:
```
#define MIN(X,Y) ((X)<(Y)?(X):(Y))
a = MIN(b,c);
```

expanded source:
```
a = ((b)<(c)?(b):(c))
```

Repeated Evaluation

- Each actual parameter is re-evaluated every time it is used

source file:
```
#define MIN(X,Y) ((X)<(Y)?(X):(Y))
a = MIN(b++,c++);
```

expanded source:
```
a = ((b++)<(c++)?(b++):(c++))
```

Capture

source file:
```
#define intswap(X,Y) {int temp=X; X=Y; Y=temp;}
int main() {
  int temp=1, b=2;
  intswap(temp,b);
  printf("%d, %d\n", temp, b);
}
```

expanded source:
```
int main() {
  int temp=1, b=2;
  {int temp= temp ;  temp = b ;  b =temp;} ;
  printf("%d, %d\n", temp, b);
}
```

Capture

- In a program fragment, any occurrence of a variable that is not statically bound is *free*
- When a fragment is moved to a different context, its free variables can become bound
- This phenomenon is called *capture*:
  - Free variables in the actuals can be captured by definitions in the macro body
  - Also, free variables in the macro body can be captured by definitions in the caller

By Name

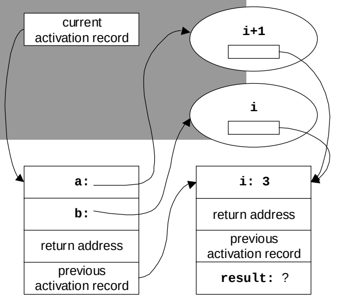
For passing parameters by name, each actual parameter is evaluated in the caller’s context, on every use of the corresponding formal parameter.

- Like macro expansion without capture
- Algol 60 and others
- Difficult to implement and nowadays unpopular

Implementing By-Name

- The actual parameter is treated like a little anonymous function
- Whenever the called method needs the value of the formal it calls the function to get it

```
void f(by-name int a, by-name int b) {
  b=5;
  b=a;
}
int g() {
  int i = 3;
  f(i+1,i);
  return i;
}
```


current activation record
i+1
i
a:
b:
return address
previous activation record
i: 3
return address
previous activation record
result: ?

When **f** is starting

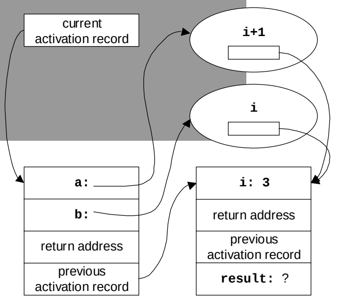
Output: 6

By Need

For passing parameters by need, each actual parameter is evaluated in the caller’s context, on the first use of the corresponding formal parameter. The value of the actual parameter is then cached, so that subsequent uses of the corresponding formal parameter do not cause reevaluation.

- Used in lazy functional languages (Haskell)
- Eliminates wasteful re-computations of by-name

```
void f(by-need int a, by-need int b) {
  b=a;
  b=a;
}
void g() {
  int i = 3;
  f(i+1,i);
  return i;
}
```


current activation record
i+1
i
a:
b:
return address
previous activation record
i: 3
return address
previous activation record
result: ?

When **f** is starting

Output:

By name: 5

By need: 4

Laziness

- All 3 passing methods (by macro, by name and by need) have in common that if the called method does not use a formal parameter the corresponding actual parameter is not evaluated.
- Some functional languages use lazy evaluation (Haskell, Miranda). Evaluate only what is necessary to give an answer.

Laziness

```
boolean andand(by-need boolean a,
               by-need boolean b) {
  if (!a) return false;
  else return b;
}

boolean g() {
  while (true) {
  }
  return true;
}

void f() {
  andand(false,g());
}
```

Here, **andand** is short-circuiting, like ML's **andalso** and Java's **&&** operators.

The method **f** will terminate without calling g.

false and g passed by need.

Exercise (exam style) Ch.18, ex. 6

```
int[] A = new int[2] ;
A[0] = 0 ;
A [1] = 2 ;
f(A[0], A[A[0]]) ;
```

```
void f(int x, int y ) {
        x = 1 ;
        y = 3 ;
}
```

For each of the following parameter passing methods, say what the final values in the array A would be, after the call to f .

a. By value
b. By reference
c. By value-result
d. By name

Exercise (exam style) Ch.18, ex. 6

```
int[] A = new int[2] ;
A[0] = 0 ;
A [1] = 2 ;
f(A[0], A[A[0]]) ;
```

```
void f(int x, int y ) {
        x = 1 ;
        y = 3 ;
}
```

For each of the following parameter passing methods, say what the final values in the array A would be, after the call to f .

a. By value	0 and 2
b. By reference	3 and 2
c. By value-result	3 and 2 or 1 and 2
d. By name	1 and 3

Summary

- How to match formals with actuals
- Seven different parameter-passing techniques
  - By value
  - By result
  - By value-result
  - By reference
  - By name
  - By need

Outline

- Subroutines and control abstraction
  - Parameters
  - ➡ Error handling

Imagine the following error situation:

Pop an empty stack

How to handle error conditions in a program?

Handling Errors Techniques

- Preconditions only
- Total definition
- Fatal errors
- Error flagging
- Throwing exceptions

1. Only Preconditions

- Document preconditions necessary to avoid errors
- Caller must ensure these are met, or explicitly check if not sure
- In Eifell language preconditions are more than just comments

```
/**
  * Pop the top int from this stack and return it.
  * This should be called only if the stack is
  * not empty.
  * @return the popped int
  */

public int pop() {
  Node n = top;
  top = n.getLink();
  return n.getData();
}
        if (s.hasMore()) x = s.pop();
        else …
```

Drawbacks

- If the caller makes a mistake, and pops an empty stack: **`NullPointerException`**
  - If that is uncaught, program crashes with an unhelpful error message
  - If caught, program relies on undocumented internals; an implementation using an array would cause a different exception

2. Total Definition

- Define some standard behavior for popping an empty stack
- Like character-by-character file I/O in C: an EOF character at the end of the file
- Like IEEE floating-point standard: NaN (not a Number) used in Java 0.0/0.0

```
/**
 * Pop the top int from this stack and return it.
 * If the stack is empty we return 0 and leave the
 * stack empty.
 * @return the popped int, or 0 if the stack is empty
 */
public int pop() {
  Node n = top;

  if (n==null) return 0;

  top = n.getLink();
  return n.getData();
}
```

Drawbacks

- Can mask important problems
- If a client pops more than it pushes, this is probably a serious bug that should be detected and fixed, not concealed

3. Fatal Errors

- The old-fashioned approach: just crash!
- Preconditions, plus decisive action
- At least this does not conceal the problem…

```
 /**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty.  If called when the stack is empty,
 * we print an error message and exit the program.
 * @return the popped int
 */
public int pop() {
  Node n = top;

  if (n==null) {
    System.out.println("Popping an empty stack!");
    System.exit(-1);
  }

  top = n.getLink();
  return n.getData();
}
```

Drawbacks

- Not an object-oriented style: an object should do things to itself, not to the rest of the program
- Inflexible: different clients may want to handle the error differently
  - Terminate
  - Clean up and terminate
  - Repair the error and continue
  - Ignore the error
  - Etc.

4. Error Flagging

- The method that detects the error can flag it somehow
  - By returning a special value (like C `malloc`)
  - By setting a global variable (like C `errno`)
  - By setting an instance variable to be checked by a method call (like C `ferror(f)`)
- Caller must explicitly test for error

```
/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty.  If called when the stack is empty,
 * we set the error flag and return an undefined
 * value.
 * @return the popped int if stack not empty
 */
public int pop() {
  Node n = top;

  if (n==null) {
    error = true;
    return 0;

  }
  top = n.getLink();
  return n.getData();
}
```

```
/**
 * Return the error flag for this stack.  The error
 * flag is set true if an empty stack is ever popped.
 * It can be reset to false by calling resetError().
 * @return the error flag
 */
public boolean getError() {
  return error;
}

/**
 * Reset the error flag.  We set it to false.
 */
public void resetError() {
  error = false;
}
```

```
/**
 * Pop the two top integers from the stack, divide
 * them, and push their integer quotient.  There
 * should be at least two integers on the stack
 * when we are called.  If not, we leave the stack
 * empty and set the error flag.
 */
public void divide() {
  int i = pop();
  int j = pop();
  if (getError()) return;
  push(i/j);
}
```

The kind of explicit error check required by an error flagging technique.

Note that `divide`'s caller may also have to check it, and its caller, and so on…

5.Throwing exceptions

```
/**
 * Pop the top int from this stack and return it.
 * @return the popped int
 * @exception EmptyStack if stack is empty
 */
public int pop() throws EmptyStack {
  Node n = top;

  if (n==null) throw new EmptyStack();

  top = n.getLink();
  return n.getData();
}
```

Advantages

- Good error message even if uncaught
- Documented part of the interface
- Error caught right away, not masked
- Caller need not explicitly check for error
- Error can be ignored or handled flexibly

Exceptions in Java

```
public class Test {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    System.out.println(i/j);
  }
}
```

What can happen when this code executes?

Options

- Option 1. Everything works fine.

```
> javac Test.java
> java Test 6 3
2
```

- Option 2. A runtime error occurs. Java **throws** automatically an **exception.** Exception = an error condition that stops the ordinary flow of computation

What happens to the exception?

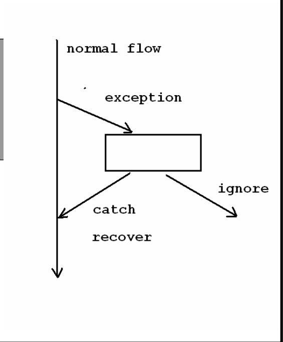
Option 2A. The program ignores it.

Java prints an error message, a core dump and terminates program.

```
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at Test.main(Test.java:3)
> java Test 6 0
Exception in thread "main"
        java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:4)
```

Option 2B. The program catches the exception and recovers from it

```
> java Test 6 3
2
> java Test 6 0
You're dividing by zero!
```

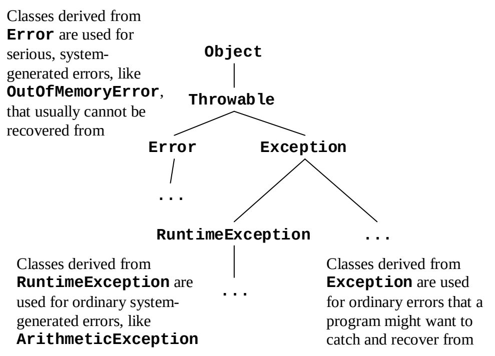


An Exception Is An Object

- The names of exceptions are class names, like **`NullPointerException`**
- Exceptions are objects of those classes
- The Java language system automatically creates an object of an exception class and *throws* it
- The program can *catch* it or not
- The program can also throw exceptions with **`throw`**
- Exceptions can be predefined or user made.

Java Predefined Exceptions

Java Exception	Code to Cause It
`NullPointerException`	`String s = null;` `s.length();`
`ArithmeticException`	`int a = 3;` `int b = 0;` `int q = a/b;`
`ArrayIndexOutOfBoundsException`	`int[] a = new int[10];` `a[10];`
`ClassCastException`	`Object x = new Integer(1);` `String s = (String) x;`
`StringIndexOutOfBoundsException`	`String s = "Hello";` `s.charAt(5);`



Classes derived from **`Error`** are used for serious, system-generated errors, like **`OutOfMemoryError`**, that usually cannot be recovered from

Classes derived from **`RuntimeException`** are used for ordinary system-generated errors, like **`ArithmeticException`**

Classes derived from **`Exception`** are used for ordinary errors that a program might want to catch and recover from

Example

```
public class Test {
  public static void main(String[] args) {
    try {
      int i = Integer.parseInt(args[0]);
      int j = Integer.parseInt(args[1]);
      System.out.println(i/j);
    }

      catch (ArithmeticException a) {
      System.out.println("You're dividing by zero!");
    }
  }
}
```

This will catch and handle any **ArithmeticException**.

Example

```
> java Test 6 3
2
> java Test 6 0
You're dividing by zero!
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at Test.main(Test.java:3)
```

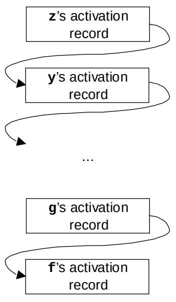
- Catch type chooses exceptions to catch:
  - **`ArithmeticException`** got zero division
  - **`RuntimeException`** would get both examples above
  - **`Throwable`** would get all possible exceptions

After The **`try`** Statement

- A **`try`** statement can be just another in a sequence of statements
- If no exception occurs in the **`try`** part, the **`catch`** part is not executed
- If no exception occurs in the **`try`** part, or if there is an exception **which is caught** in the **`catch`** part, execution continues with the statement following the **`try`** statement

Long-Distance Throws

- If **`z`** throws an exception it does not catch, **`z`**'s activation stops…
- …then **`y`** gets a chance to catch it; if it doesn't, **`y`**'s activation stops…
- …and so on all the way back to **`f`**



Multiple catch Example

```
public static void main(String[] args) {
  try {
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    System.out.println(i/j);
  }
  catch (ArithmeticException a) {
    System.out.println("You're dividing by zero!");
  }

  catch (ArrayIndexOutOfBoundsException a) {
    System.out.println("Requires two parameters.");
  }
}
```

Overlapping Catch Parts

- If an exception from the **`try`** part matches more than one of the **`catch`** parts, only the first matching **`catch`** part is executed
- A common pattern: **`catch`** parts for specific cases first, and a more general one at the end
- Note that Java does not allow unreachable **`catch`** parts, or unreachable code in general

Overlapping catch Example

```
public static void main(String[] args) {
  try {
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    System.out.println(i/j);
  }
  catch (ArithmeticException a) {
    System.out.println("You're dividing by zero!");
  }
  catch (ArrayIndexOutOfBoundsException a) {
    System.out.println("Requires two parameters.");
  }
  catch (RuntimeException a) {
    System.out.println("Runtime exception.");
  }
}
```

Clean-up operations

- In Java **`finally`** part is used for cleanup operations that must to be executed no matter what

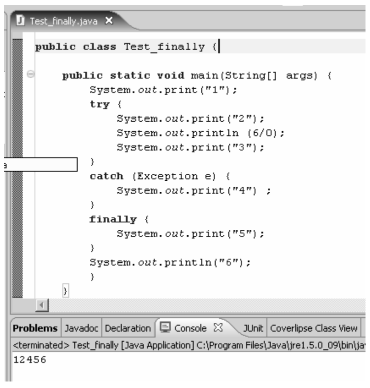
```
file.open();
try {
  workWith(file);
}
finally {
  file.close();
}
```

Finally

- Finally executes immediately before normal exit, immediately after executing catch clause or before other escapes in try because of otherexecptions.

Example

```
System.out.print("1");
try {
System.out.print("2");
System.out.println (6/0);
System.out.print("3");
}
catch (Exception e) {
System.out.print("4") ;
}
finally {
System.out.print("5");
}
System.out.println("6");
```



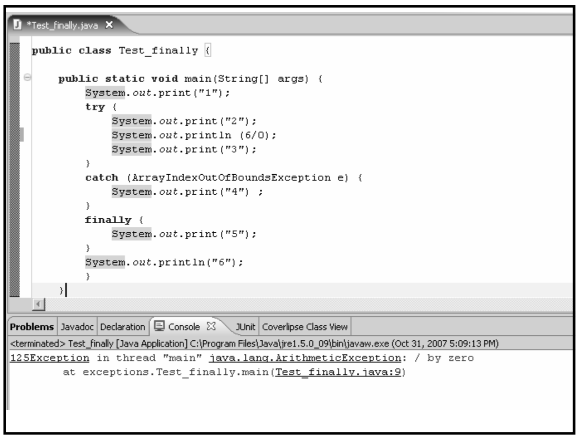
```
public class Test_finally {

    public static void main(String[] args) {
        System.out.print("1");
        try {
            System.out.print("2");
            System.out.println (6/0);
            System.out.print("3");
        }
        catch (Exception e) {
            System.out.print("4") ;
        }
        finally {
            System.out.print("5");
        }
        System.out.println("6");
        }
    }
```

```
<terminated> Test_finally [Java Application] C:\Program Files\Java\jre1.5.0_09\bin\ja
12456
```

Example

```
System.out.print("1");
try {
System.out.print("2");
System.out.println (6/0);
System.out.print("3");
}
catch (ArrayIndexOutOfBoundsException e) {
System.out.print("4") ;
}
finally {
System.out.print("5");
}
System.out.println("6");
```



```
public class Test_finally {

    public static void main(String[] args) {
        System.out.print("1");
        try {
            System.out.print("2");
            System.out.println (6/0);
            System.out.print("3");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.print("4") ;
        }
        finally {
            System.out.print("5");
        }
        System.out.println("6");
        }
    }
```

```
<terminated> Test_finally [Java Application] C:\Program Files\Java\jre1.5.0_09\bin\javaw.exe (Oct 31, 2007 5:09:13 PM)
125Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exceptions.Test_finally.main(Test_finally.java:9)
```

Summary

- There are many ways to handle errors. Exceptions are one of them
- In Java exceptions are objects of class `Throwable`
- Exceptions can be thrown by the language system automatically or by the program (custom exceptions)
- The program may catch or not the thrown exceptions