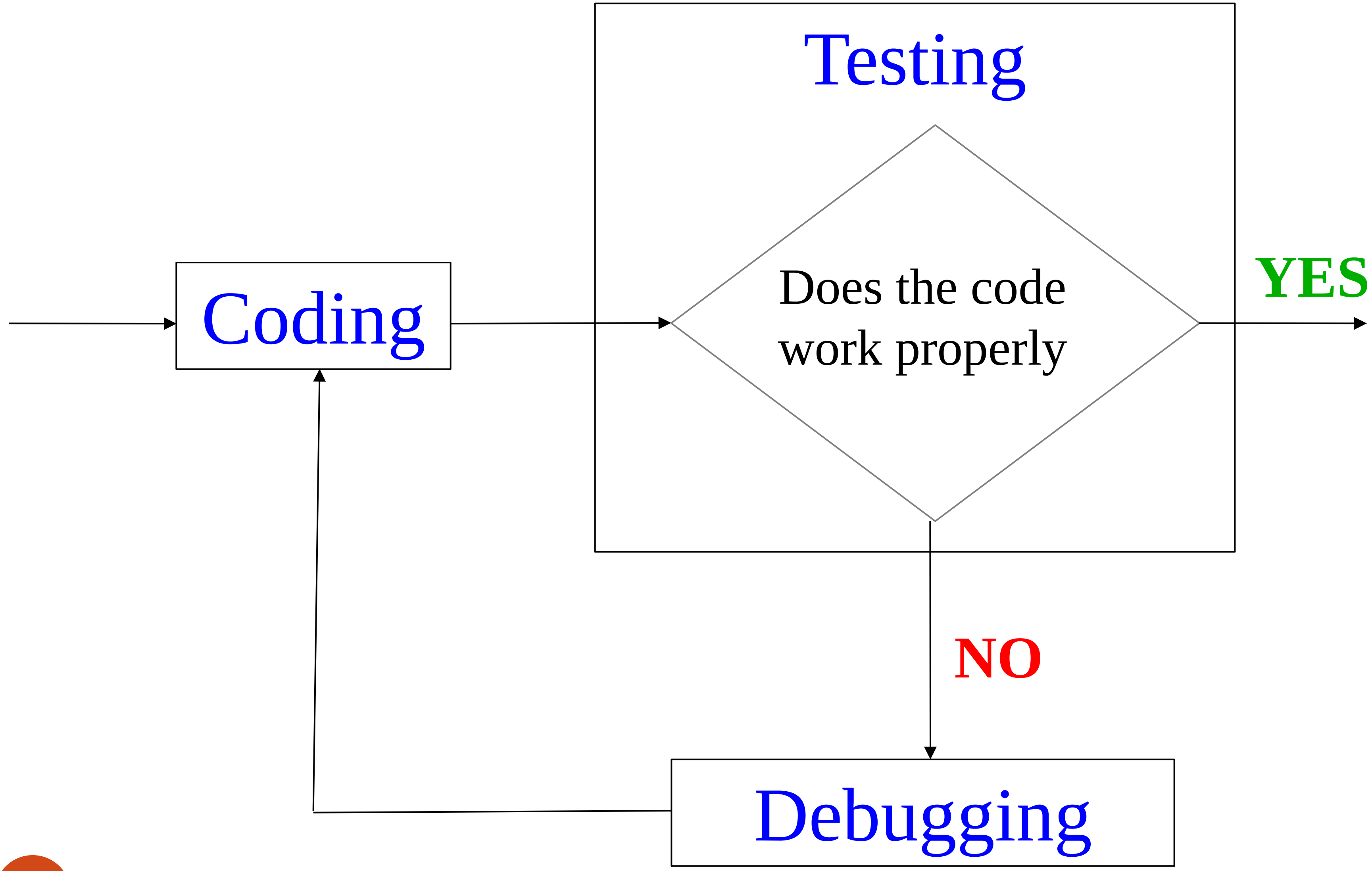
Designing with Exceptions

CSE219, Computer Science III

Stony Brook University

http://www.cs.stonybrook.edu/~cse219

Testing vs. Debugging



Testing

Coding

Does the code work properly

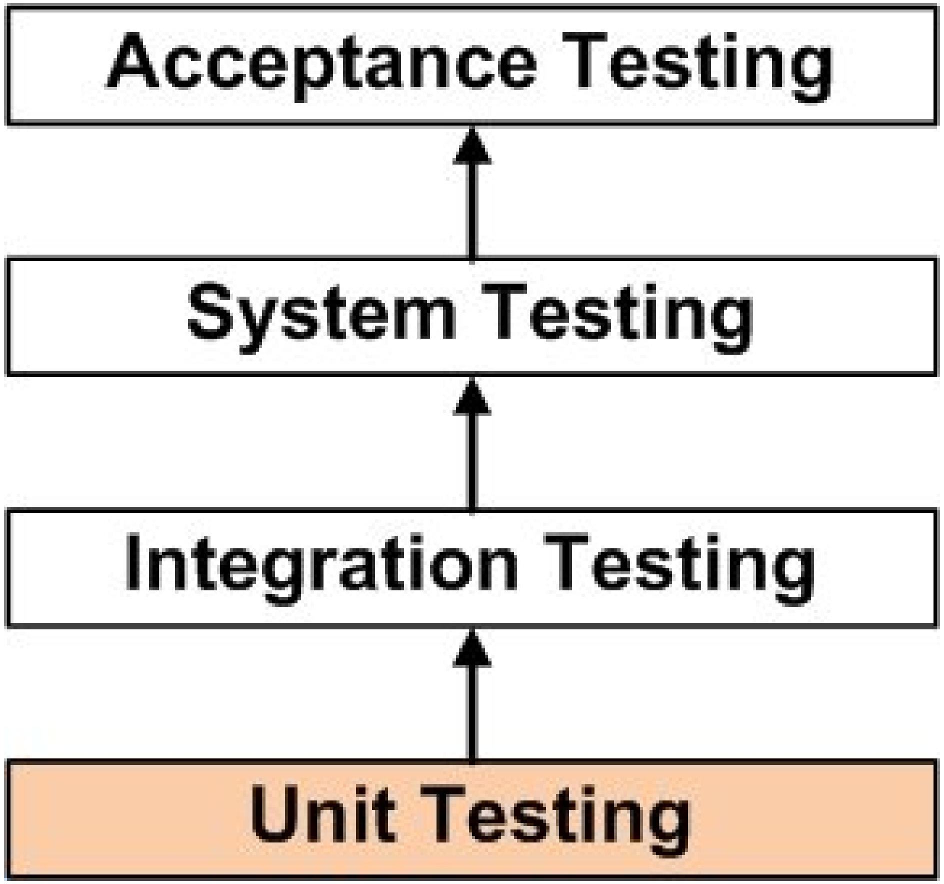
YES

NO

Debugging

Testing

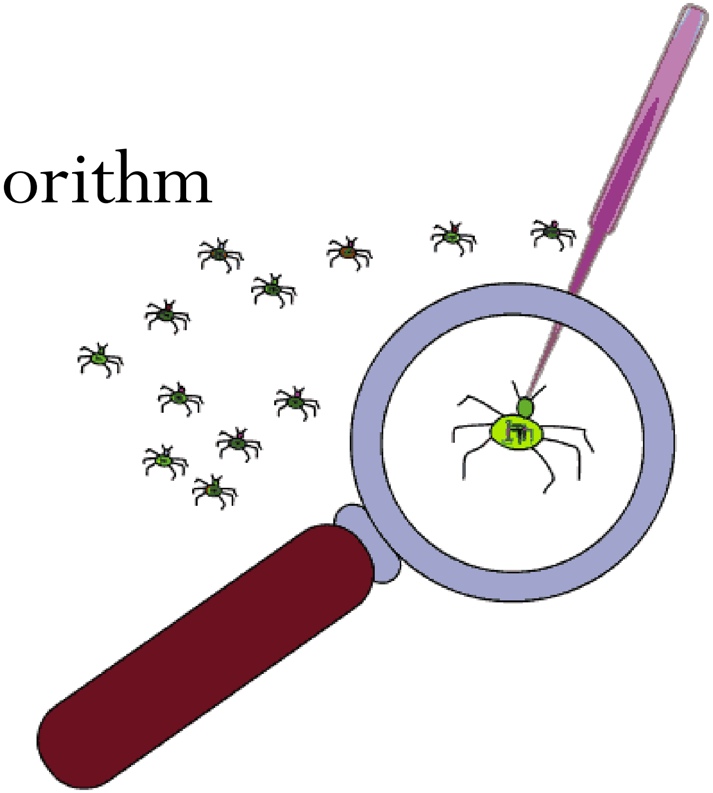
- Tells us when something is wrong
  - not how to fix it



Debugging

- Process of understanding and correcting errors
- First locate the problem
  - find line of your code that produces initial problem
- Then address the algorithm
  - correct implementation of algorithm

    OR
  - change algorithm



Debugging is an important skill

- Become proficient ASAP
- Why?
  - Reveal bugs that are not otherwise evident
    - like infinite loops
- Don't design to debug
  - Don't rely on debugging to write your code
    - Try to define and implement correct algorithms
  - fast debugging << correct algorithm implementation

Debugging Strategy

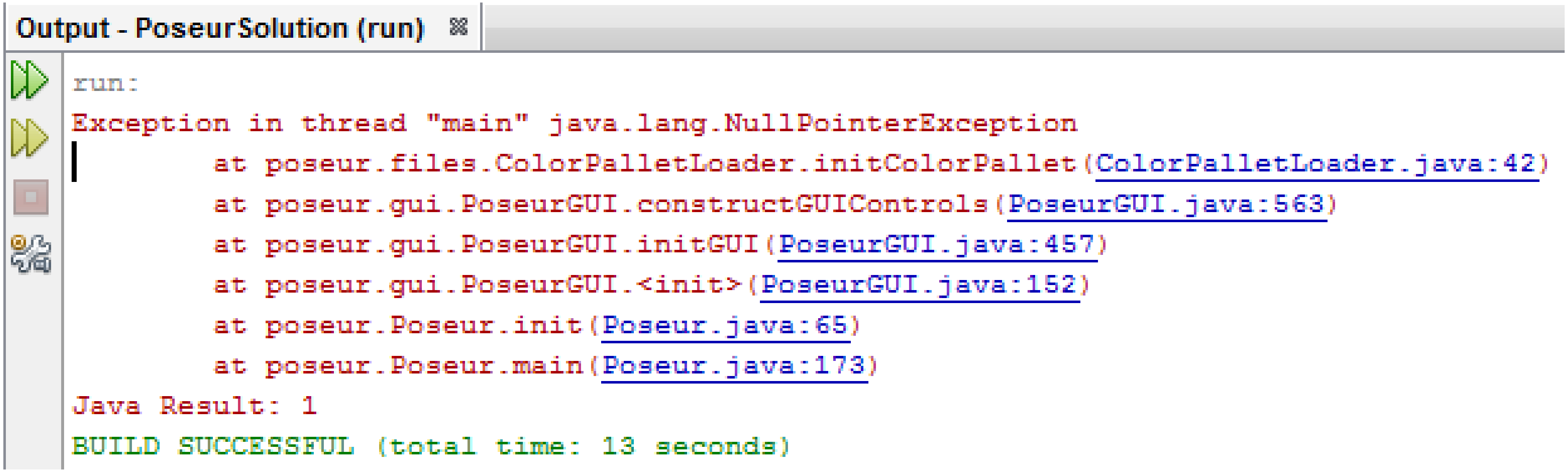
- When you know a bug exists for a particular case
  - Determine in which class the error originates
  - Determine in which method the error originates
  - Determine on which line of code the error originates
- Knowing where the problem originates is half the battle
- Reproducing an error helps

Common Bugs Revealed by Debugging

- Un-constructed Objects
- Improper Iteration
- Un-initialized Variables
- Missing Implementations
- Failing to reinitialize a variable in loop
- Incomplete Changes

Not all errors are created equal

- On difficulty scale:
  - syntax errors << runtime errors << logical errors
- Note:
  - runtime errors may be due to logical errors



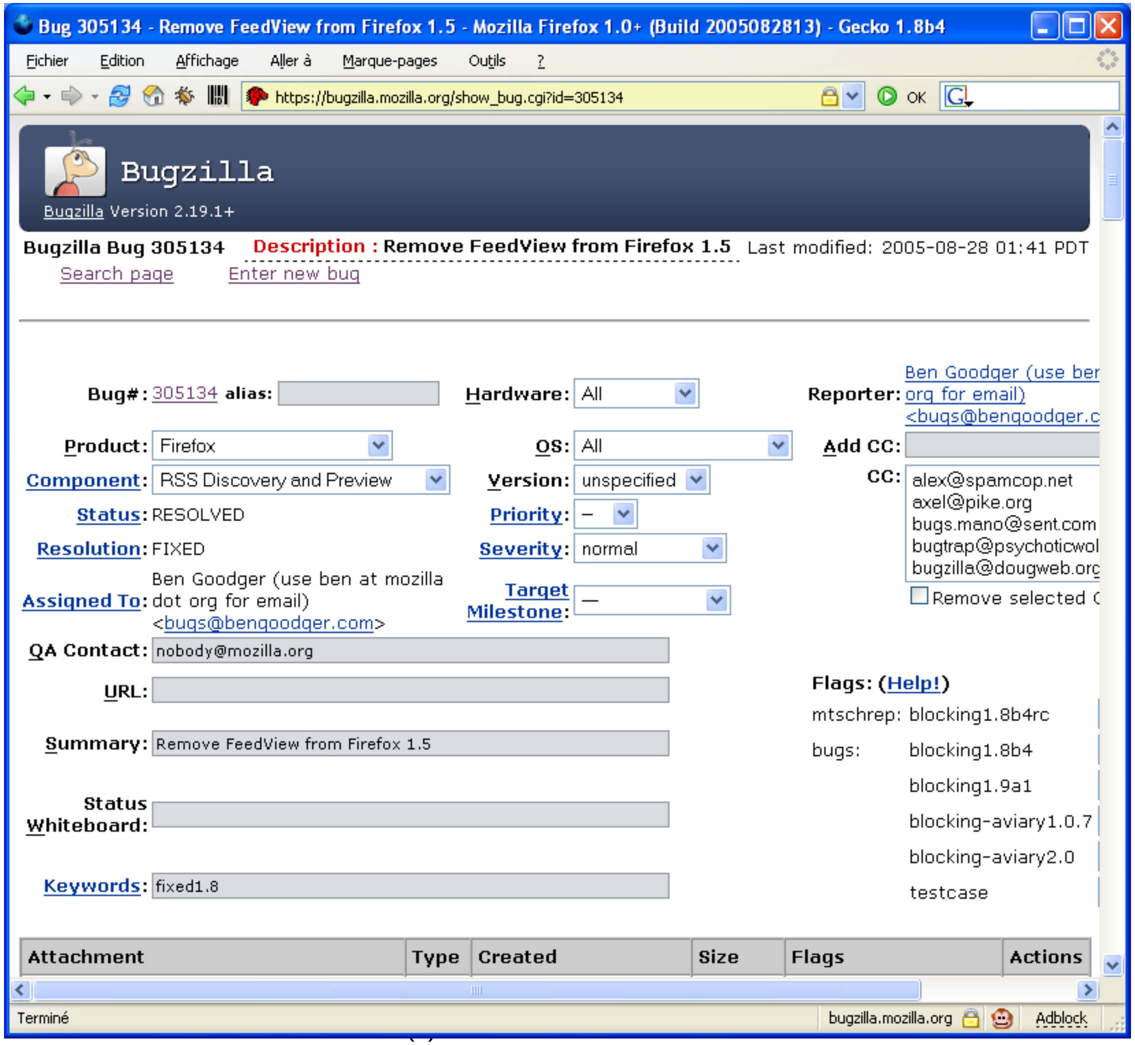
```
run:
Exception in thread "main" java.lang.NullPointerException
        at poseur.files.ColorPalletLoader.initColorPallet(ColorPalletLoader.java:42)
        at poseur.gui.PoseurGUI.constructGUIControls(PoseurGUI.java:563)
        at poseur.gui.PoseurGUI.initGUI(PoseurGUI.java:457)
        at poseur.gui.PoseurGUI.<init>(PoseurGUI.java:152)
        at poseur.Poseur.init(Poseur.java:65)
        at poseur.Poseur.main(Poseur.java:173)
Java Result: 1
BUILD SUCCESSFUL (total time: 13 seconds)
```

Plan to Debugging

- Assumption:
  - every program will contain faults
  - no programmer gets it right the first time
- So?
  - Design, write, & document your programs in ways that will make them easier to test & debug
- How?
  - write well-documented modular code
  - avoid “I'll fix this later” approach

Professionals use tools

- Even for tracking bugs (e.g., Bugzilla)


Bug 305134 - Remove FeedView from Firefox 1.5 - Mozilla Firefox 1.0+ (Build 2005082813) - Gecko 1.8b4
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
https://bugzilla.mozilla.org/show_bug.cgi?id=305134
OK
Bugzilla
Bugzilla Version 2.19.1+
Terminé
bugzilla.mozilla.org
Adblock

Bugzilla Bug 305134 **Description :** Remove FeedView from Firefox 1.5 Last modified: 2005-08-28 01:41 PDT

Search page Enter new bug

Bug#:	305134 **alias:**	**Hardware:**	All	**Reporter:**	Ben Goodger (use ben org for email) <bugs@bengoodger.c
Product:	Firefox	**OS:**	All	**Add CC:**	
Component:	RSS Discovery and Preview	**Version:**	unspecified	**CC:**	alex@spamcop.net axel@pike.org bugs.mano@sent.com bugtrap@psychoticwol bugzilla@dougweb.org
Status:	RESOLVED	**Priority:**	--		
Resolution:	FIXED	**Severity:**	normal		
Assigned To:	Ben Goodger (use ben at mozilla dot org for email) <bugs@bengoodger.com>	**Target Milestone:**	---		☐ Remove selected C
QA Contact:	nobody@mozilla.org				
URL:				**Flags: (Help!)**	
Summary:	Remove FeedView from Firefox 1.5			mtschrep:	blocking1.8b4rc
				bugs:	blocking1.8b4
					blocking1.9a1
Status Whiteboard:					blocking-aviary1.0.7
					blocking-aviary2.0
Keywords:	fixed1.8				testcase

Attachment	Type	Created	Size	Flags	Actions

Debugging by Brute Force

- I.e. the print statement
  - display contents of select variables
  - display benchmarks of program progress
    - i.e. Is this line of code reached?

```
System.out.println(“Before Foo”);
foo();
System.out.println(“After Foo”);
```

- Advantage:
  - easy to implement

Disadvantages of print Approach

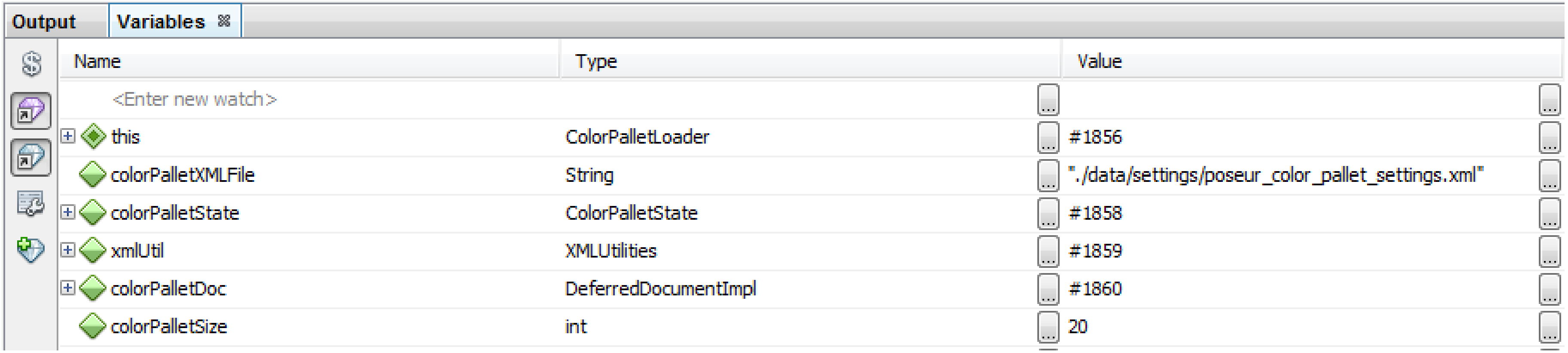
- Makes a mess of code
- Hit-or-miss
- Can't identify certain types of problems
- Not easy to use for:
  - Large-scale programs
  - Graphical programs
  - Web apps
  - Mobile apps

Debugging by Brute Force Example

```
private static boolean debug = true;
...
public int calculate (int y, int z) {
  int x;
  x = mystery(y);
  if (debug) {
     System.out.println(“DEBUG: x = ” + x
          + “ y = ” + y);
  }
  x += mystery(z);
  return x;
}
```

Debugging by IDE

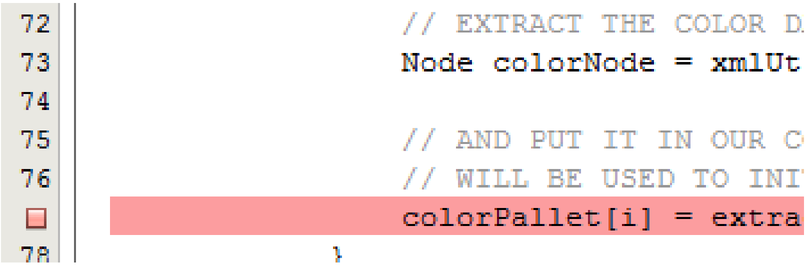
- All modern IDEs provide:
  - examination of the contents of variables
  - setting and removing of breakpoints
  - query and search commands
  - single-step execution through a program
  - examination of different threads of execution



Name	Type	Value
<Enter new watch>		
this	ColorPalletLoader	#1856
colorPalletXMLFile	String	"./data/settings/poseur_color_pallet_settings.xml"
colorPalletState	ColorPalletState	#1858
xmlUtil	XMLUtilities	#1859
colorPalletDoc	DeferredDocumentImpl	#1860
colorPalletSize	int	20

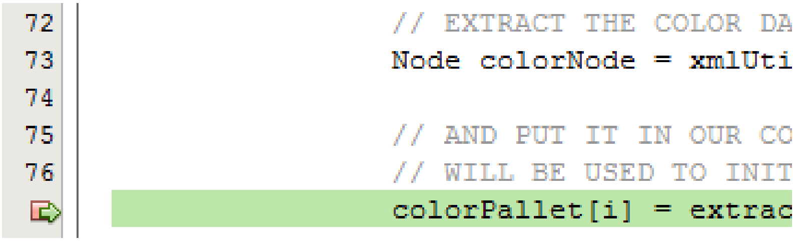
NetBeans Debugger

- Similar to other IDE debuggers
  - eclipse, Visual Studio, etc.
- Set Breakpoints
  - place where debugger will stop
- Walk through code via:
  - Stop
  - Pause
  - Continue
  - Step Over
  - Step Over Expression
  - Step Into
  - Step Out



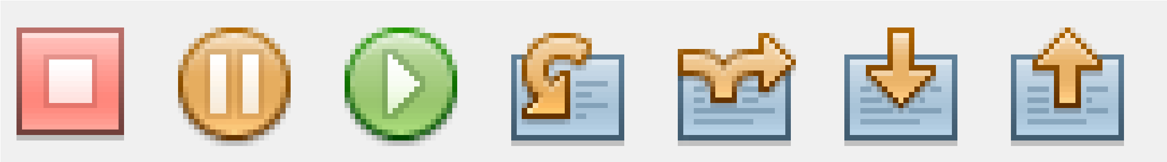
```
// EXTRACT THE COLOR D
Node colorNode = xmlUt

// AND PUT IT IN OUR C
// WILL BE USED TO INI
colorPallet[i] = extra
}
```



```
// EXTRACT THE COLOR DA
Node colorNode = xmlUti

// AND PUT IT IN OUR CO
// WILL BE USED TO INIT
colorPallet[i] = extrac
```



Robust Programs

- Methods have domain (arguments) & range (results)
- Total methods – behavior is defined for all inputs in the method domain
  - By definition these are robust methods
- Partial methods can lead to programs that are not robust
- Robust program continues to behave reasonably even in the presence of errors
  - If an error occurs, robust programs behave in a well-defined way. Either:
    - Providing some approximation of its behavior in the absence of an error = ***graceful degradation***

    OR
    - Halt with a meaningful error message without crashing or damage to permanent data or software systems

Exceptions

- Allow the flow of control to move from the location of an error to an error handler
  - Better than returning -1?
    - Treats errors differently from normal results
    - Forces the programmer to deal with these errors
- Types of errors:
  - User input errors
  - Device errors
  - Physical limitations
  - Code errors
- An exception is an abstraction
  - allows us to handle errors in a more general way

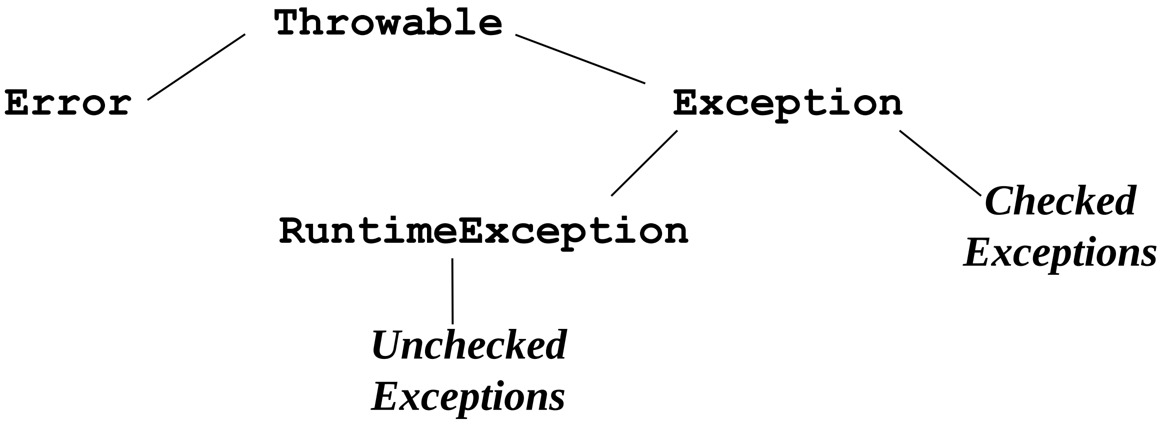
Exceptions/Errors in Java

- An exception may be thrown because:
  - A method is called that throws a *checked* exception.
    - **`FileNotFoundException, IOException`**
  - A method is called that detects an error and explicitly throws a *checked* exception.
    - Create your own class that **`extends Exception`**.
  - A method throws an *unchecked* exception due to a programming error (i.e. a run-time logical error).
    - **`ArithmeticException,NullPointerException`**
  - An internal error occurs in the Java Virtual Machine (JVM) or runtime library.
    - e.g. **`VirtualMachineError, OutOfMemoryError`**

Method Design w/ Exceptions

- Throw an exception when a method's preconditions are not met
  - As well as any other error condition found in the method
- Throw different types of exceptions for different types of problems
- Specify detailed information about the reason for the exception in the Exception message
- Provide a *specification* of all exceptions possibly thrown inside a method

Exceptions in Java



Throwable

Error

Exception

RuntimeException

Checked Exceptions

Unchecked Exceptions

- A method throwing a *checked* exception must declare the exception in the header via throws
- A method throwing an *unchecked* exception does not have to declare the exception in its header
  - but it is advisable to do so!
    - also, make sure your specification explains the conditions that generate each exception

Handling Exceptions

- An exception is handled in two ways:
  - Enclose the method call that can cause an exception in a **`try`** block.
    - Use a **`catch`** block to handle the possible exception.
  - Pass the exception back to the current method's caller.
    - Java automatically passes the exception to the method's caller if:
      - the exception type of one of its supertypes is listed in the method's header (in a **`throws`** clause)
      - the exception type is unchecked
- Again! Make sure that any exception your code raises is listed in the header and is described in the method's specification.

Tips on Using Exceptions

- Too much exception handling will slow your code down dramatically.
- Exception handling is not supposed to replace a simple test by an application.
- Robust GUIs should check input from users before processing information.
- Exceptions serve to protect the methods & classes that throw them,
  - *Defensive programming*: writing each procedure to defend itself against errors.

Tips on Using Exceptions

- Do not micromanage exceptions
  - Example: Read a string and convert it to an int

```
try {
     line = inFile.readLine();
} catch (IOException e) {
     System.out.println(e);
}
try {
     num = Integer.parseInt(line);
} catch (NumberFormatException e) {
     System.out.println(e);
}
```

Put both exceptions into a single catch!

Tips on Using Exceptions

- Continue example:

```
try {
    line = inFile.readLine();
    num = Integer.parseInt(line);
}catch (IOException e) {
    System.out.println(e);
}catch (NumberFormatException e) {
    System.out.println(e);
}
```

And separate normal processing from error handling.

Tips on Using Exceptions

- Do not squelch/suppress/ignore exceptions.
  - Example: Popping off a stack with 100 elements.

```
sum = 0;
for (i=1; i <= 100; i++){
     try {
          sum += s.pop();
     }catch(EmptyStackException e){
     } // squelched!
}
```

  - Logical errors can be completely missed if exceptions are ignored!

Reflecting is Good

- Method A calls method B, which throws an exception, rather than passing the exception:
  - The caller A explicitly catches the exception from B and throws a different type of exception.
    - Example: Find the min of an array.
      - Method begins by trying to get the element in position 0.
    - If the array is empty, **`IndexOutOfBoundsException`** is thrown.
    - The min method may catch this and return **`EmptyArrayException`**.
  - Why would we want to do this?
    - Turn vague exceptions into more relevant ones!
    - Turn unchecked exceptions into checked ones!

```
public static int min(int[] a) throws
EmptyArrayException {
      try{
            int min = a[0];
            …
      }catch(IndexOutOfBoundsException e)
            throw new EmptyArrayException();
      }
}
```

Masking

- Method A calls method B, which throws an exception.
  - The caller A explicitly catches and handles the exception and continues with the normal flow
  - Any method calling A is none the wiser
    - Example: Sorting an array.
      - Method tries to get element in position 0.
      - If the array is empty, the array is already sorted (by definition).
      - Method catches **`IndexOutOfBoundsException`** and masks it.

Design Issues with Exceptions

- When should one use them?
- Checked or unchecked?
- Use existing Exception classes or make your own?

When Do We Use Exceptions?

- Exceptions should be used to prevent data (static or instance variables) from reaching an illegal state
  - Make a partial method more like a total method
- Exceptions may be avoided (by returning an `int` error code) if a method is used only locally
  - Ex: `private` helper methods
- Use exceptions for exceptional situations
- Special Java rule for overriding:
  - If you override a method, the subclass method cannot throw more checked exceptions than the superclass that you replace.

Use checked or unchecked?

- Always use checked exceptions!
- Why?
  - let other programmers (and yourself) be aware of potential errors
  - make them anticipate these errors
  - make them handle these errors as they see fit
- Many exceptions in the JDK are unchecked. Why?
  - It would clutter the code (example: having a try block for every indexed array, division or object use).

Programmer vs. User

- Unchecked exceptions occurring are generally the fault of the programmer
- Checked exceptions occurring may be the fault of the user/system/Internet access

Testing and debugging in large projects

- Testing using frameworks:
  - JUnit
    - Unit testing framework for the Java programming language
      - Testing individual components
    - Used in regression testing

```
import org.junit.*;… TestSuite suite= new TestSuite();  suite.addTest( new Test(…))
```

  - Apache Log4J
    - Logging results of applications
      - Also used in debugging Web applications
    - Properties stored in property file *log4j.properties:*

```
log = /usr/home/log4j
log4j.rootLogger = DEBUG, FILE
```

    - Use: import org.apache.log4j.Logger; … static Logger log = Logger.getLogger(log4jExample.class.getName()); … log.debug("this is an debug message");