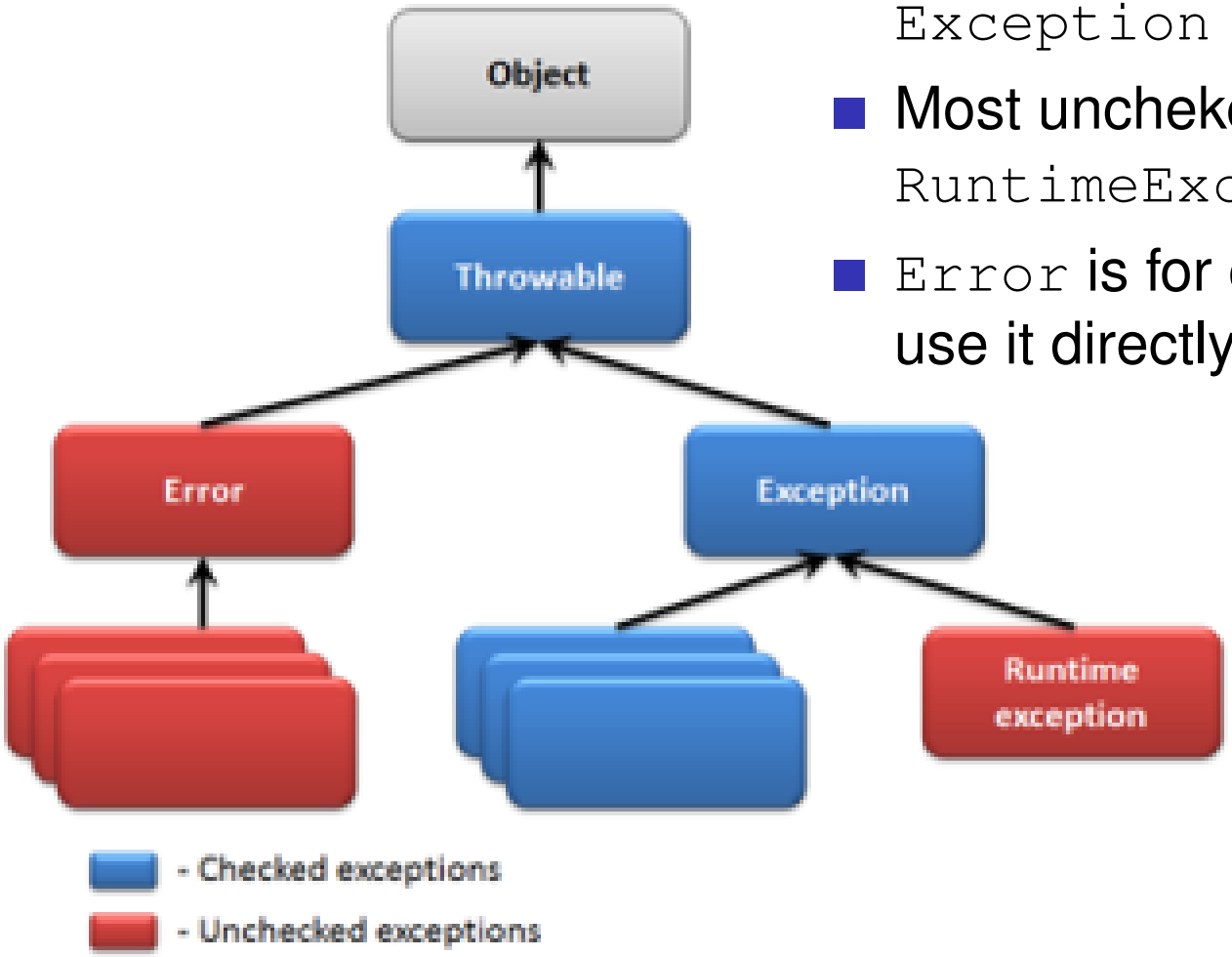
# Programming with Exceptions

Christopher Simpkins
chris.simpkins@gatech.edu

# Programming with Exceptions

- Understanding Java's exception class hierarchy
- Understanding how exceptions work
- Principles for programming with exceptions
- Writing and using your own exception classes

# Java's Exception Hierarchy




- Most (checked) exceptions will subclass `Exception`
- Most uncheked exceptions will subclass `RuntimeException`
- `Error` is for compiler hackers. Don't use it directly.

# Throwing Exceptions is a Control Flow Mechanism

What does this code print?

```
public class Wee {

    static void bar() throws Throwable {
        throw new Throwable("Wee!");
    }

    static void foo() throws Throwable {
        bar();
        System.out.println("Foo!");
    }

    public static void main(String[] args) {
        try {
            foo();
        } catch (Throwable t) {
            System.out.println(t.getMessage());
        }
        System.out.println("I'm still running.");
    }
}
```

# Principles of Exception Handling

- Don't try to handle coding errors.
- Prefer exceptions from the standard library to creating your own.
- Name exceptions after the problem (not the thrower).
- Wrap exceptions when crossing an abstraction boundary.
- Store useful information in exception objects.
- Handle exceptions close to their origins, but ...
  - Assign exception-handling responsibility to objects that can handle the exceptions.
  - Don't "eat" exceptions (at the very least, log the exception).
  - If you can't handle an excepotin sensibly, propagate it ("when in doubt, throw it out").

See `http://wirfs-brock.com/PDFs/towards_xcptn_hndling.pdf` for more details. We'll just touch on a few of these here.

# Applying Exception Handling Principles

What's wrong with this constructor?

```
public class Company {
    private ArrayList<Employee> employees;

    public Company(String employeeDataFile) {
        employees = new ArrayList<Employee>(10);
        try {
            initFromFile(new File(employeeDataFile));
        } catch (FileNotFoundException e) {
            System.out.println("Employee data file not found.");
        } catch (ParseException e) {
            System.out.println("Malformed data file: "+e.getMessage();
        } catch (Exception e) {
            System.out.println("Exception occurred: "+e.getMessage());
        }
    }
    //...
}
```

# Writing and Using Your Own Exceptions

Define your own exception classes by subclassing `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).

```
public class MyException extends Exception {

    public MyException(String msg) {
        super(msg);
    }
}
```

And use them just like any other exception:

```
if (checkProblem()) {
    throw new MyException("Oops!");
}
```

But remember: in most cases there is an Exception class in the standard library that you can use. Don't write your own exception classes unless you really need to.

# Use The Most Specific Applicable Exception

Recall our `Company` constructor:

```
try {
    initFromFile(new File(employeeDataFile));
} catch (FileNotFoundException e) {
    //...
} catch (ParseException e) {
    //...
} catch (Exception e) {
    //...
}
```

With separate exceptions we can take more specific actions, e.g.:

- We can tell the user to check for the right file (FileNotFoundException).
- We can tell the user that the data file is malformed (ParseException).

# Final Thoughts

- Use exceptions for their intended purpose: separating your core logic from the code that handles exceptional conditions.
- Use exceptions judiciously (not too many).
- Think about how you handle exceptions:
  - have sound reasons for propagating exceptions you propagate
  - have sound reasons for catching exceptions where you catch them
  - recover if you can
  - store information in your exceptions to aid in debugging or error recovery by the user