Principles of Software Construction: Objects, Design and Concurrency

Exceptions and Classes (cont.), Packages, and Inheritance

15-214

toad

Fall 2012

Jonathan Aldrich **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

© 2012 C Garrod, J Aldrich, and W Scherlis

Administrivia

- Homework 0 – due tonight
  - To confirm your homework submission, svn checkout in a new location
- Homework 1 coming soon

institute for SOFTWARE RESEARCH

Key object concepts from last Thursday

- Inside an object
  - Kinds of members: Fields, Methods, Constructors
  - Visibility from the outside: hiding the members
  - The keyword *this*
- Interfaces and the management of expectations
  - Java interfaces
  - Introduction to types
- Objects and the heap
  - Method dispatch
- Objects and identity
  - Equals vs. ==
- Exceptions

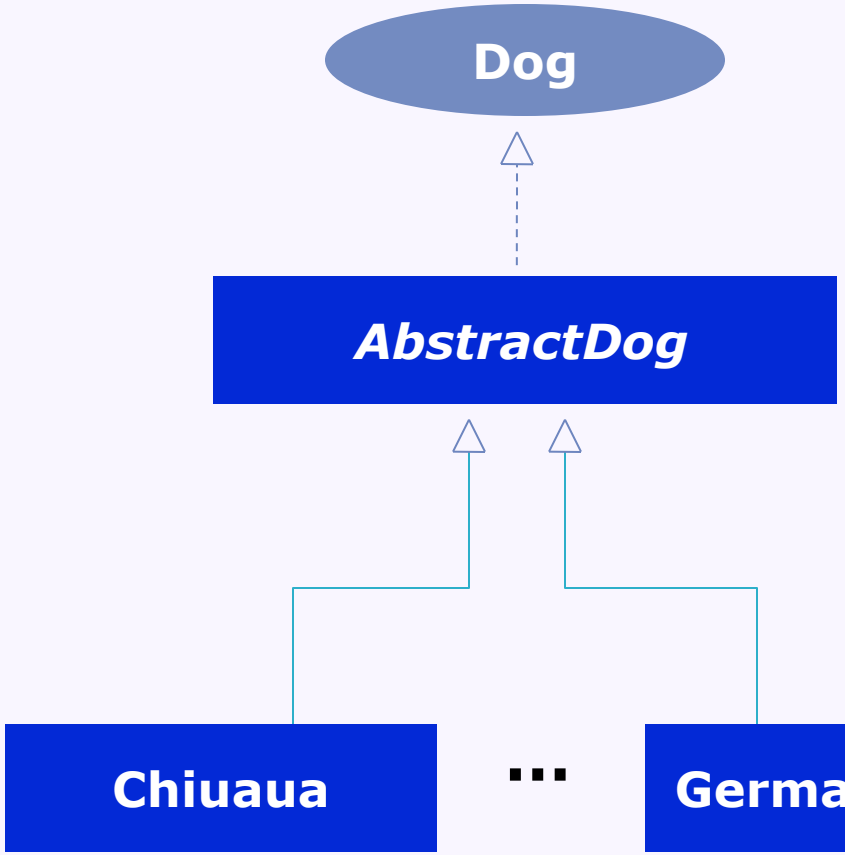
institute for SOFTWARE RESEARCH

Key object concepts for today

- Exceptions (continued)
- Classes, revisited
  - Objects vs. classes
  - Null references
  - Mutability
    - Abstract vs. implementation
  - Static fields and methods
- Packages
  - Name and visibility management
  - Qualified names
- Inheritance
  - Reuse
  - Visibility: protected and default
  - Method dispatch, revisited

institute for SOFTWARE RESEARCH

A glimpse ahead: Inheritance, class hierarchy



"parent"
or
"superclass"

"child"
or
"subclass"

Exceptions

- Exceptions notify the caller of an exceptional circumstance (usually operation failure)
- Semantics
  - An exception propagates *up the function-call stack* until `main()` is reached or until the exception is caught
- Sources of exceptions:
  - Programmatically throwing an exception
  - Exceptions thrown by the Java runtime

institute for SOFTWARE RESEARCH

Benefits of exceptions

- Provide high-level summary of error and stack trace
  - Compare: core dumped in C
- Can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Can optionally recover from failure
  - Compare: calling `System.exit()`
- Improve code structure
  - Separate routine operations from error-handling
- Allow consistent clean-up in both normal and exceptional operation

institute for SOFTWARE RESEARCH

Exceptions improve code structure

- Compare to this (fake) code fragment:

```
FileInputStream fIn = new FileInputStream(filename);
if (fIN == null) {
  switch (errno) {
  case _ENOFILE:
     System.err.println("File not found: " + …);
     return -1;
  default:
     System.err.println("Something else bad happened: " + …);
     return -1;
  }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
  System.err.println("Unknown internal error.");
  return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
  System.err.println("Error reading binary data from file");
  return -1;
}  // I didn't have enough room to close the file.  Oh well.
return i;
```

Catching exceptions, control flow with `finally`

```
try {
   dangerousOperation();
   System.out.println("All is well!");
} catch (MildException e) {
   recover();
} catch (DeadlyException e) {
   System.err.println("Whoops!  Don't die.");
   revive();
} finally {
    // put code here that we always want to run
    // at the end of the try/catch block
}
```

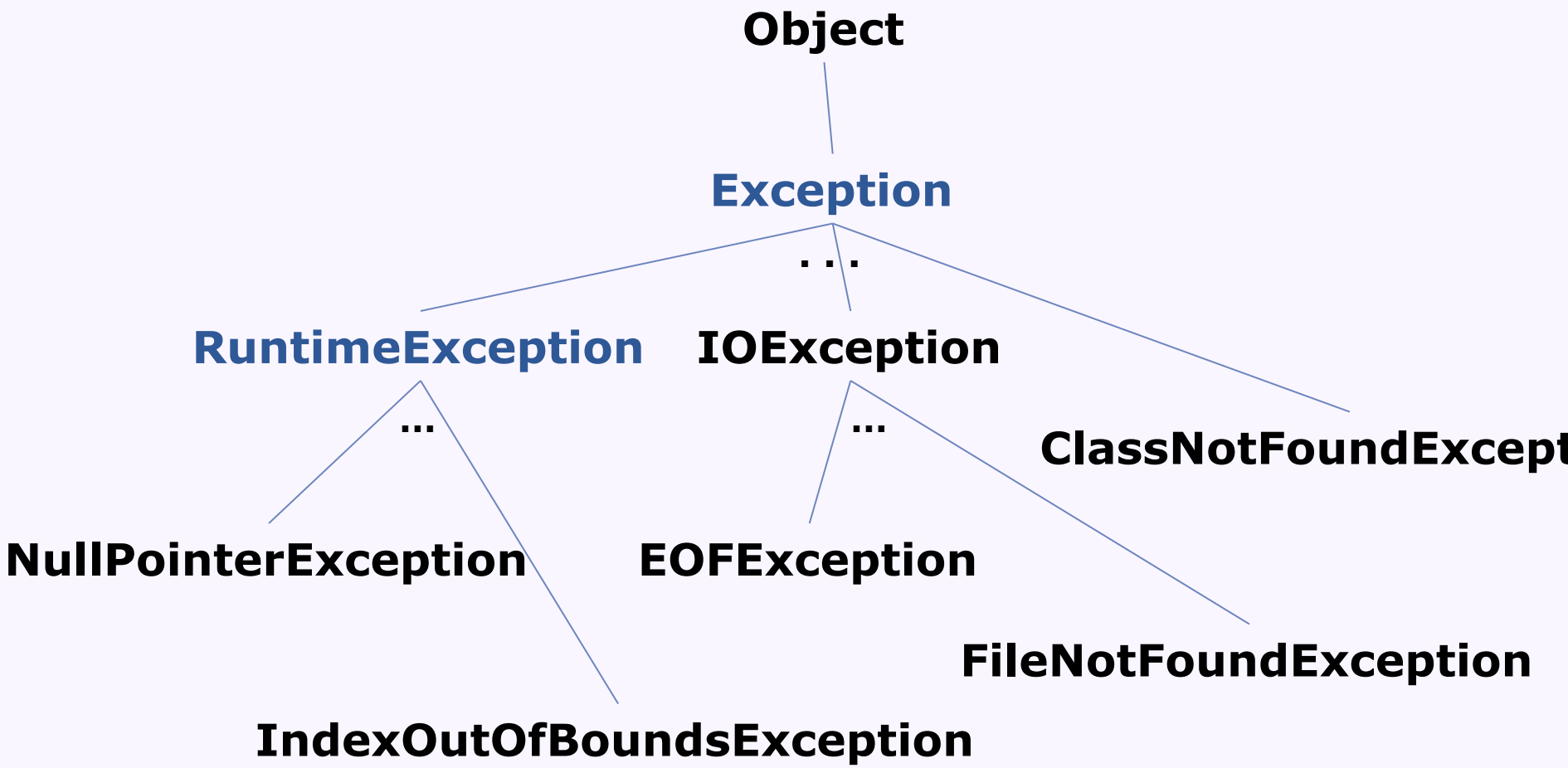
Throwing exceptions

- Exceptions are classes that *extend* the `java.lang.Exception` class
- Basic use:

```
if (someErrorBlahBlahBlah) {
    throw new MyCustomException("Blah blah blah");
}
```

- See `IllegalBowlingScoreException` and `ReadBowlingScore` for an example

The exception hierarchy



Checked and unchecked exceptions

- Unchecked exception: any subclass of `RuntimeException`
  - Indicates an error which is highly unlikely and/or typically unrecoverable

- Checked exception: any subclass of `Exception` but not `RuntimeException`
  - Indicates an error that every caller should be aware of and explicitly decide to handle or pass on

institute for SOFTWARE RESEARCH

Guidelines for using exceptions

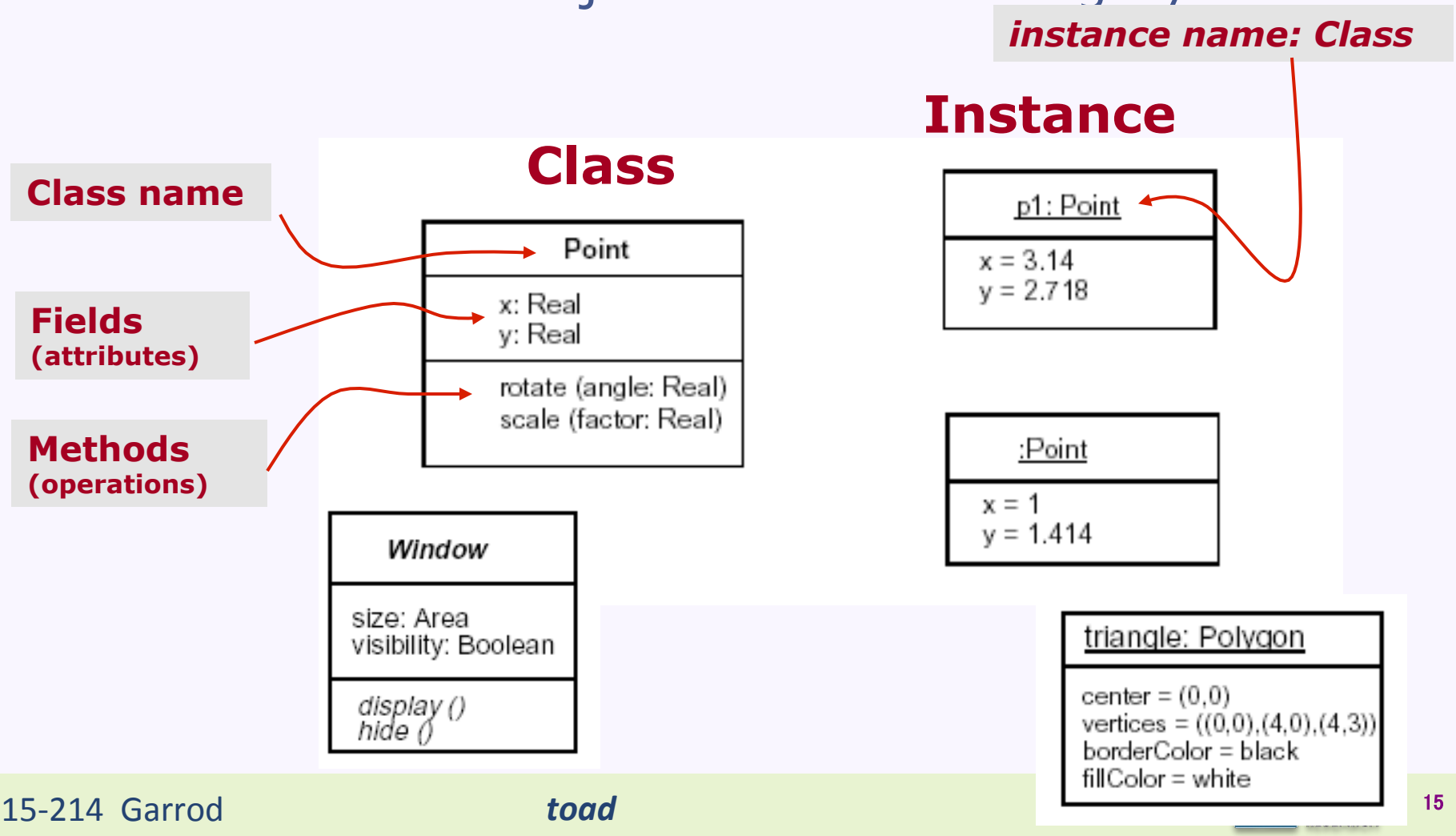
- Catch and handle all checked exceptions
  - Unless there is no good way to do so, in which case you should pass them on to your caller or throw a `RuntimeException`
- Use runtime exceptions for programming errors
  - If you receive bad input, throw a subclass of `RuntimeException`
- Other good practices
  - Do not catch an exception without (at least somewhat) handling the error
  - When you throw an exception, describe the error
  - If you re-throw an exception, always include the original exception as the cause

Key object concepts for today

- Exceptions (continued)
- Classes, revisited
  - Objects vs. classes
  - Null references
  - Mutability
    - Abstract vs. implementation
  - Static fields and methods
- Packages
  - Name and visibility management
  - Qualified names
- Inheritance
  - Reuse
  - Visibility: protected and default
  - Method dispatch, revisited

Relating objects and classes

- A *class*: a category of entities
- An *instance*: an object within the category



Null object references

- An object data field can be null
  - Uninitialized or explicitly set to null
  - Refers to no heap data
- An attempt to dereference a null reference is an error
  - `NullPointerException`

```
String alice = "Alice";
String bob = null;
if (bob.equals(alice)) {
    …
}
```

- Advice:
  - Avoid relying upon null references when possible
    - e.g., see the `EmptyIntList`

Static members

- The idea of static
  - State and actions associated with an entire class (as opposed to being associated with individual objects
- Examples
  - A simple `Counter` example
  - The `main` method – why is this static?
  - Some `String` examples (coming up!)
    - `valueOf`

institute for SOFTWARE RESEARCH

Mutability and immutability

- Data is *mutable* if it can change over time. Otherwise it is *immutable*.
- Data is *abstract immutable* if its private internal representation is mutable but the data is immutable from an external client's perspective
  - e.g., a Java `String`

Confusion alert: "static" and "immutable" are unrelated concepts here!

institute for SOFTWARE RESEARCH

Java `Strings`, an (approximate) look inside

- Fields

```
char[]    value
int       len
int       offset
int       hash
```

- Quick tour:
  - Representation of a string
  - Static `.valueOf`
  - String objects are abstract immutable
    - Internal representation is mutable: `hash`
    - How `.equals` is implemented
  - Why a private constructor?
    - How `.subString` is implemented
  - The many shapes of `new String( … )`
    - Method dispatch

institute for SOFTWARE RESEARCH

Key object concepts for today

- Exceptions (continued)
- Classes, revisited
  - Objects vs. classes
  - Null references
  - Mutability
    - Abstract vs. implementation
  - Static fields and methods
- Packages
  - Name and visibility management
  - Qualified names
- Inheritance
  - Reuse
  - Visibility: protected and default
  - Method dispatch, revisited

Coming Thursday