Exception Handling and First-class Functions

Corky Cartwright

Vivek Sarkar

Department of Computer Science

Rice University

Errors and Exceptions in Java

- In Java, a common supertype, Throwable, is used to encompass all error values and exception values.

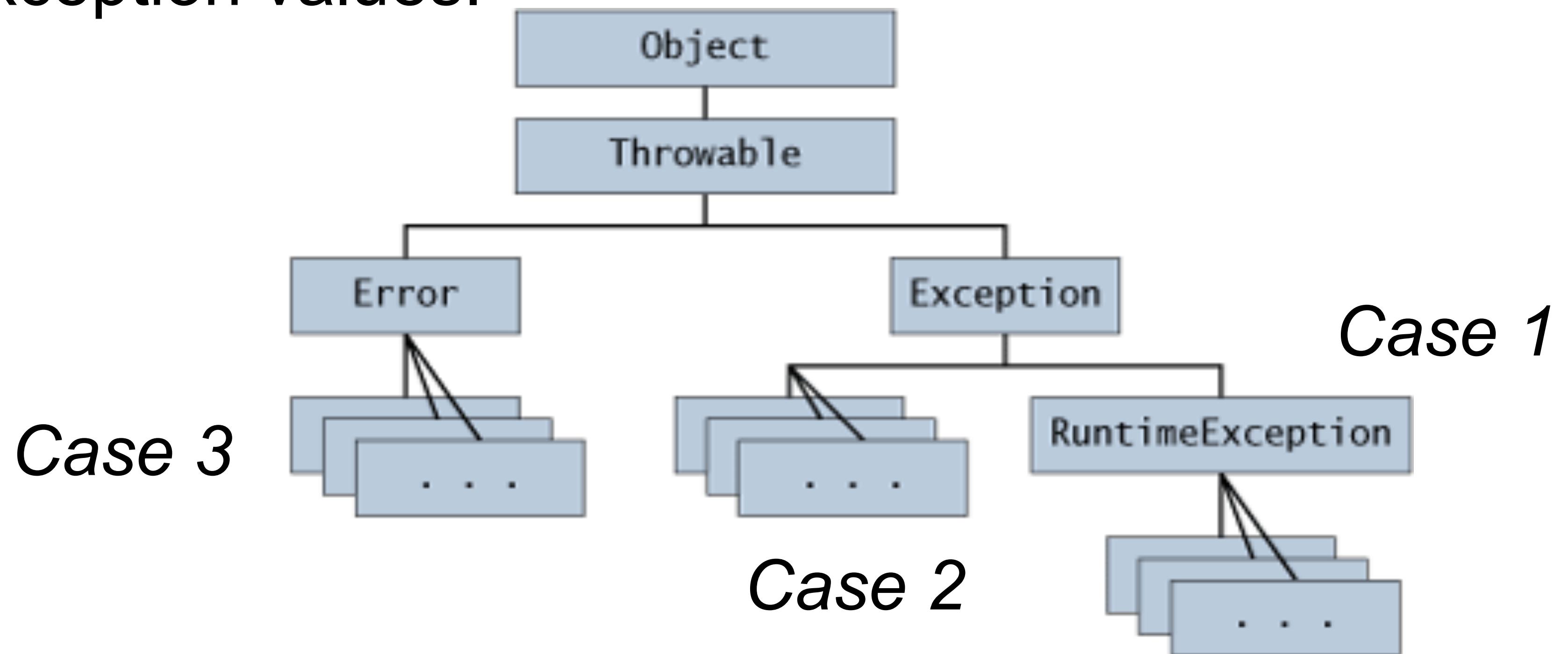


Figure source: http://java.sun.com/docs/books/tutorial/essential/exceptions/throwing.html

Case 1: RuntimeException

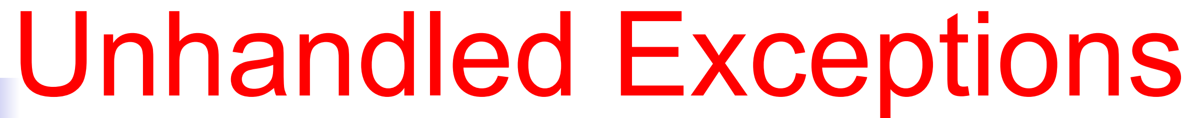
- Used for error conditions that a program may want to handle, but are not explicitly part of a method's contract e.g.,
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException (e.g., divide by zero)`
  - `NegativeArraySizeException`
  - `ArrayStoreException`
  - **`ClassCastException`**
  - **`IllegalArgumentException`**
- We will primarily use RuntimeException (Case 1) in this course except when the use of a library dictates the use of Case 2 or Case 3

Example

Execution of method foo() in class T1 throws an `ArithmeticException` when x = 0

```
class T1 {
  int x;
  . . .
  int foo() {
    return 100 / x;
  }
}
```

Unhandled Exceptions



- An Unhandled Exception results in program exit with a stack trace e.g.,

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at T1.foo(T1.java:50)

. . .
```

- The line numbers in the stack trace can help you locate the source of the error

Handled Exceptions

- The programmer has the option of handling exceptions in Java with a try-catch statement e.g.,

```
class T1 {
  int x;
  . . .
  int foo() {
    int n;
    try { n = 100 / x; } // scope of exception handler
    catch (ArithmeticException e)
      {n = Integer.MAX_VALUE;} // handler for arith exceptions
    return n;
  } }
```

Exception Propagation

- Exceptions are propagated up the call chain until a handler is found; if none an error message is printed on the console

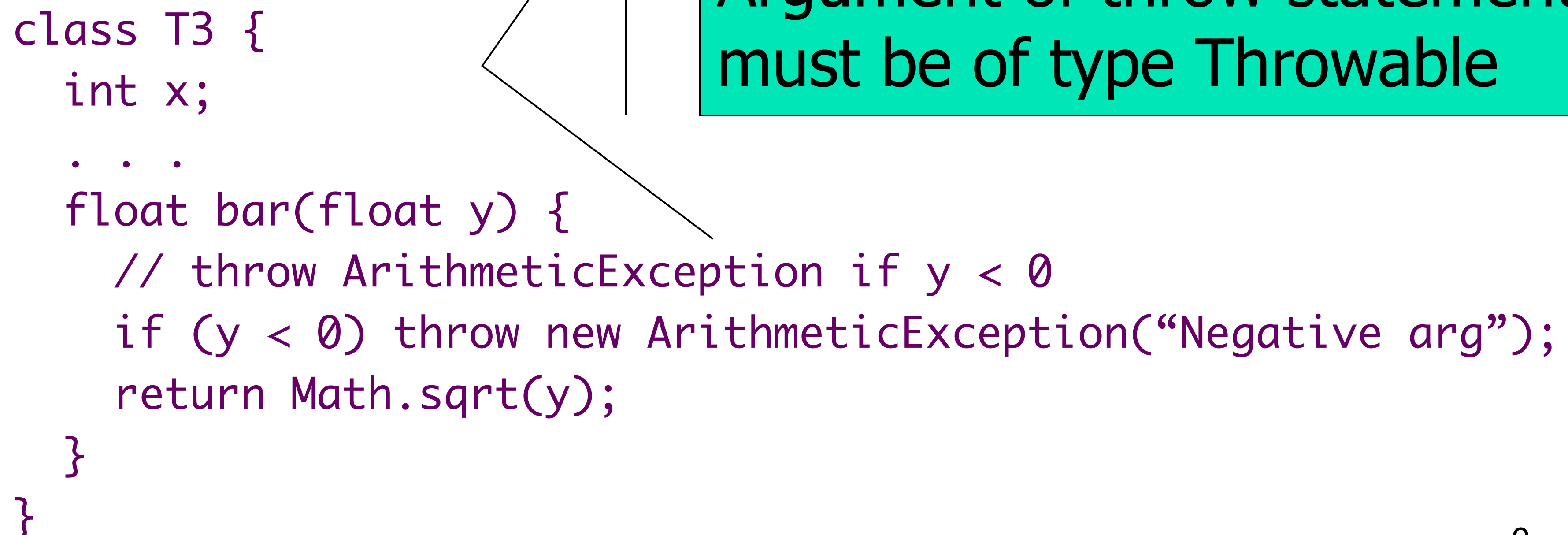
```
class T2 {
  int x;
  int baz() { return 100 / x; }
  int foo() {
    int n;
    try { baz(); } // scope of exception handler
    catch (ArithmeticException e)
      {n = Integer.MAX_VALUE;} // handler for arith exceptions
    return n;
  } }
```

Food for Thought

- What would you have to do to propagate errors up a call chain in a language that did not have support for exception handling?
- It is possible to convert any Java program into one that never prints an exception on the console. How?

Throwing Exceptions Explicitly

- The programmer also has the option of throwing instances of RuntimeException for user-defined errors e.g.,




```
class T3 {
  int x;
  . . .
  float bar(float y) {
    // throw ArithmeticException if y < 0
    if (y < 0) throw new ArithmeticException("Negative arg");
    return Math.sqrt(y);
  }
}
```

Exception Objects

- In Java, exceptions are conventional objects, and can be created by expressions of the form

```
new <exception-class>(<arg1>, ..., <argn>)
```

- Examples

```
throw new IllegalArgumentException
   ("max applied to an empty list")

throw new java.util.NoSuchElementException
   ("no more elements")
```

Type Casts and ClassCastException

- Java supports type casts (coercions) for cases when the declared or inferred type of an expression is weaker than what is required for a particular computation
- `(<type>) <expr>` simply converts the type of `<expr>` to `<type>` for type-checking purposes. If the value of `<expr>` does not have type `<type>`, the computation throws a `ClassCastException`.
- If the cast needs to be performed repeatedly, it is also possible to assign `<expr>` to a new variable declared to be of `<type>`
- **Example**: consider the `merge` method on `IntList` for today's homework (HW7) written using the conventional Scheme solution.

merge Example

```
abstract class IntList {
  IntList cons(Comparable n) { return new ConsIntList(n, this); }
  abstract IntList merge(IntList other);
}

class EmptyIntList extends IntList {
  static EmptyIntList ONLY = new EmptyIntList();
  private EmptyIntList() { }
  IntList merge(IntList other) { return other;}
}

class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList merge(IntList other) {
    if (other == EmptyIntList.ONLY) return this;
    ConsIntList o = (ConsIntList) other; // cast operation
    if (first < o.first()) return rest.merge(o).cons(first);
    else return merge(o.rest()).cons(o.first());
  }
}
```

Cast needed because first() can only be invoked on ConsIntList

Casting vs. Compiler Type-Checking

- The type-checking in the Java compiler disallows casts
  `(<type>) <expr>`
  where `<type>` is an object type and the static type of `<expr>` and `<type>` do not overlap (other than `null`)

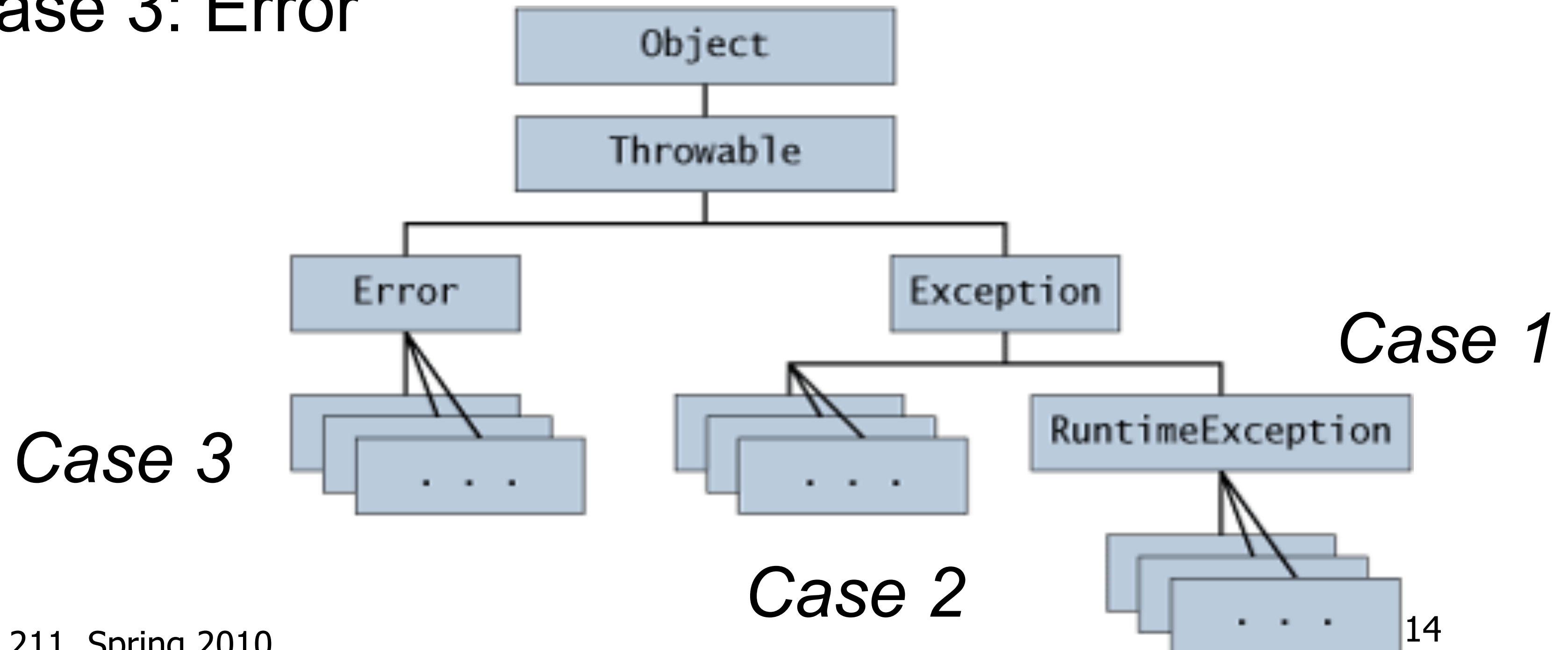
- For example

```
EmptyIntList e = new EmptyIntList();
ConsIntList o = (ConsIntList) e;
```

will result in a compile-time error

Cases 2 and 3

- Case 2: subtype of Exception, but not a subtype of RuntimeException (also called “checked exceptions”)
- Case 3: Error



Case 2: Checked Exceptions

- Used for error conditions that a program may want to handle, and that are also explicitly part of a method's contract in the throws clausee.g.,
  - `void foo() throws MyException { . . . }`

- The Java compiler enforces the following rules on checked exceptions
  - Every method that throws a checked exception must advertise it in the throws clause in its method definition (contract)
  - Every method that calls a method that advertises a checked exception must either handle that exception (with try and catch) or must in turn advertise that exception in its own throws clause.

Case 3: Errors

- Subtypes of Error are used to identify error conditions that normal programs (including all your programs!) are not expected to handle
- One direct subtype of Error is VirtualMachineError, which in turn includes the following direct subtypes
  - InternalError
  - OutOfMemoryError
  - StackOverflowError
  - UnknownError
- A VirtualMachineError is “thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating”