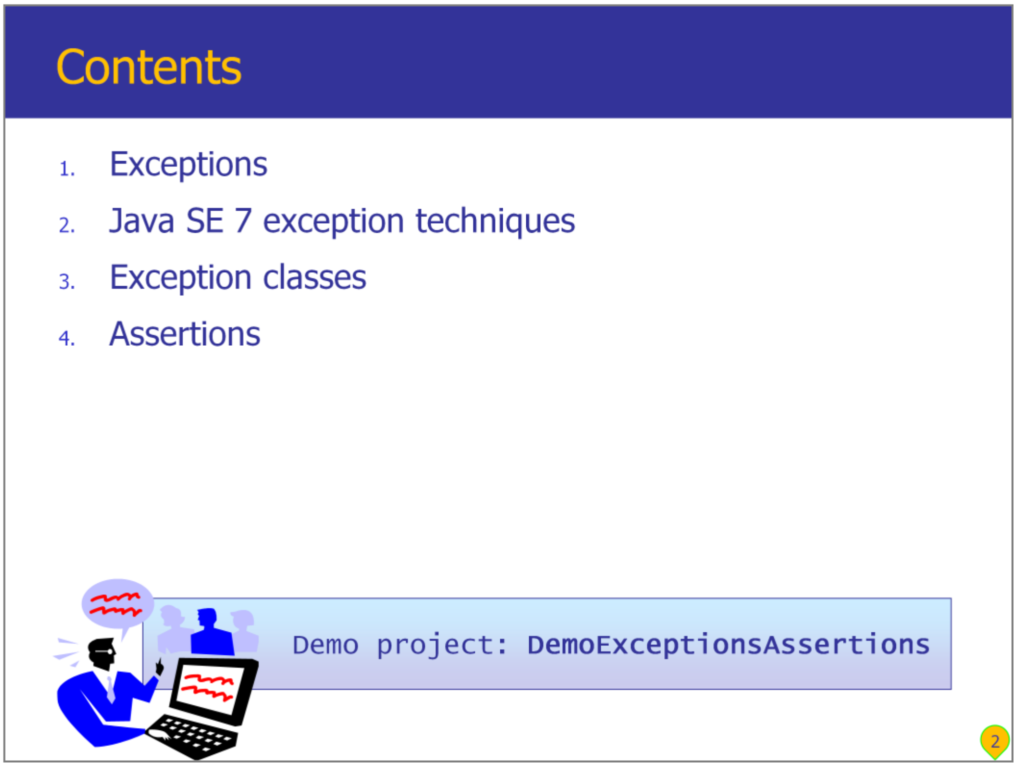
Exceptions and Assertions

Bad things can happen when an application is running. The way Java applications signal error conditions is by throwing exceptions. This chapter describes the mechanism and explains how to throw and catch exceptions in your own code.

Assertions are like mini if-tests where you can assert (or ensure) that a particular condition is true in your code. Assertions are very useful during testing, where you call a method in your code and assert that it returns the correct result.



Contents

Demo project: `DemoExceptionsAssertions`

2

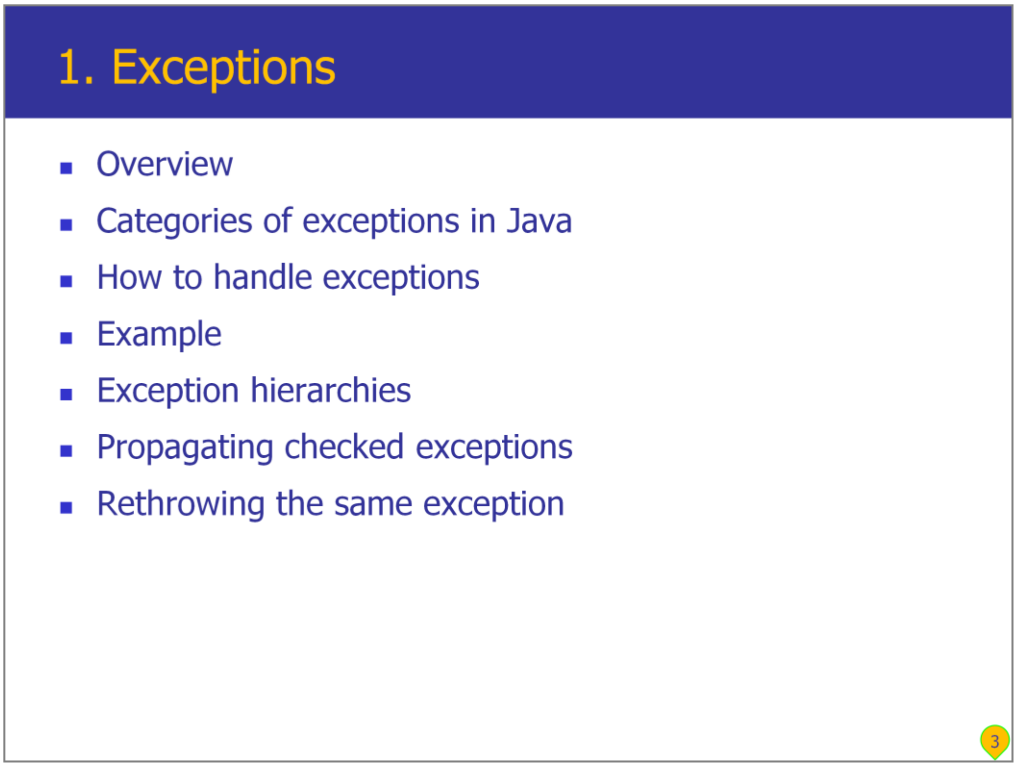
Section 1 explains the overall exception mechanism in Java, and goes into details on how to throw and catch exceptions. This is the main syntax section in this chapter.

Section 2 looks at some new exception-related techniques that were introduced into the Java language in Java SE version 7.

Section 3 describes some of the common standard exception classes in the Java SE library, and gives some hints and tips on how to handle them.

Section 4 introduces assertions. We explain what they are, how they work, and how to use them in your code.

The demos for chapter are located in the `DemoExceptionsAssertions` project.

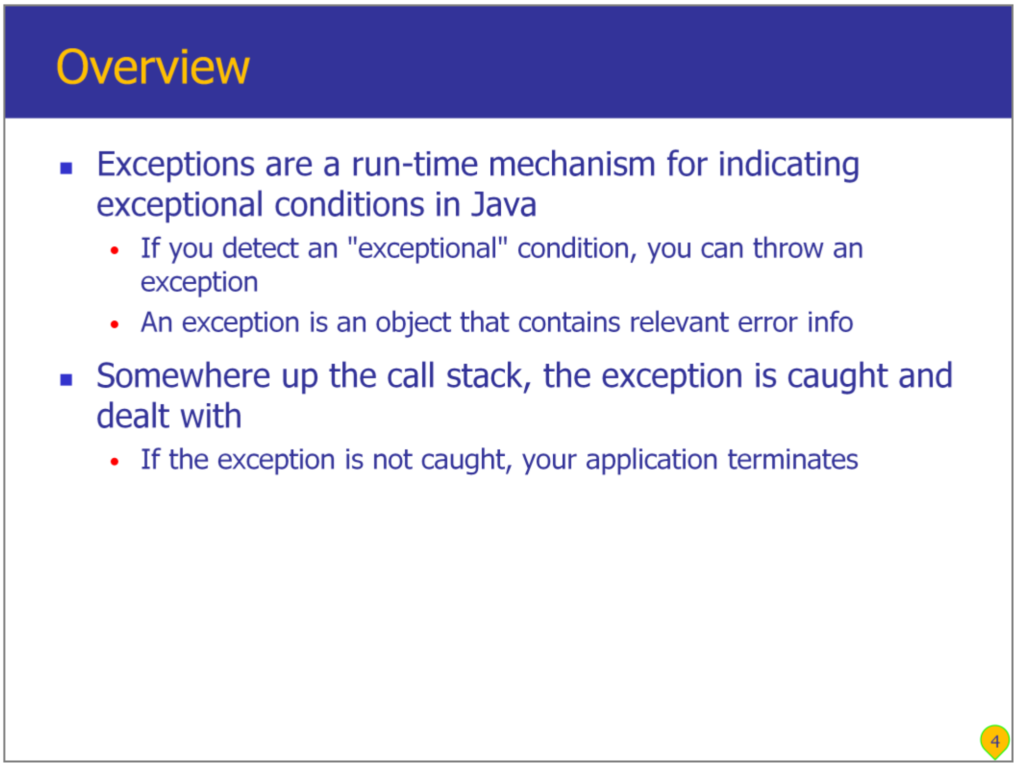


1. Exceptions

- Overview
- Categories of exceptions in Java
- How to handle exceptions
- Example
- Exception hierarchies
- Propagating checked exceptions
- Rethrowing the same exception

3

This is the main section in this chapter. In this section we'll explain what exceptions are, and go into the syntax details on how they work in Java.



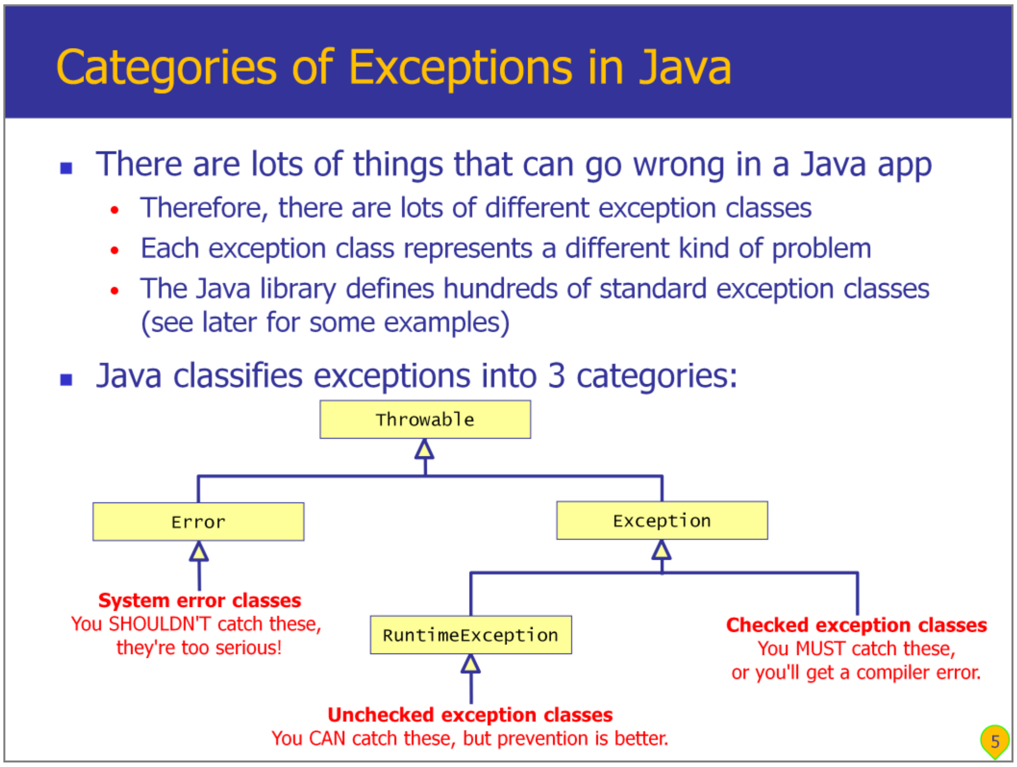
Overview

- Exceptions are a run-time mechanism for indicating exceptional conditions in Java
  - If you detect an "exceptional" condition, you can throw an exception
  - An exception is an object that contains relevant error info
- Somewhere up the call stack, the exception is caught and dealt with
  - If the exception is not caught, your application terminates

4

If you detect a problematic situation in your application, you have to think about what to do next. You could return an error value from the method (e.g. `null`, `-1`, etc.) but the calling method could just ignore the return value. A more robust approach is to throw an exception…

An exception is an object that contains some contextual information about the problem at hand. When you throw an exception, your current method terminates immediately, and the exception object is passed up the call chain to the calling method. The calling method can catch the exception and handle the problem. If the calling method doesn't catch the exception, the exception passes up to the next layer in the call stack, and so on until it eventually gets to `main()`. If `main()` doesn't catch the exception, your program terminates.



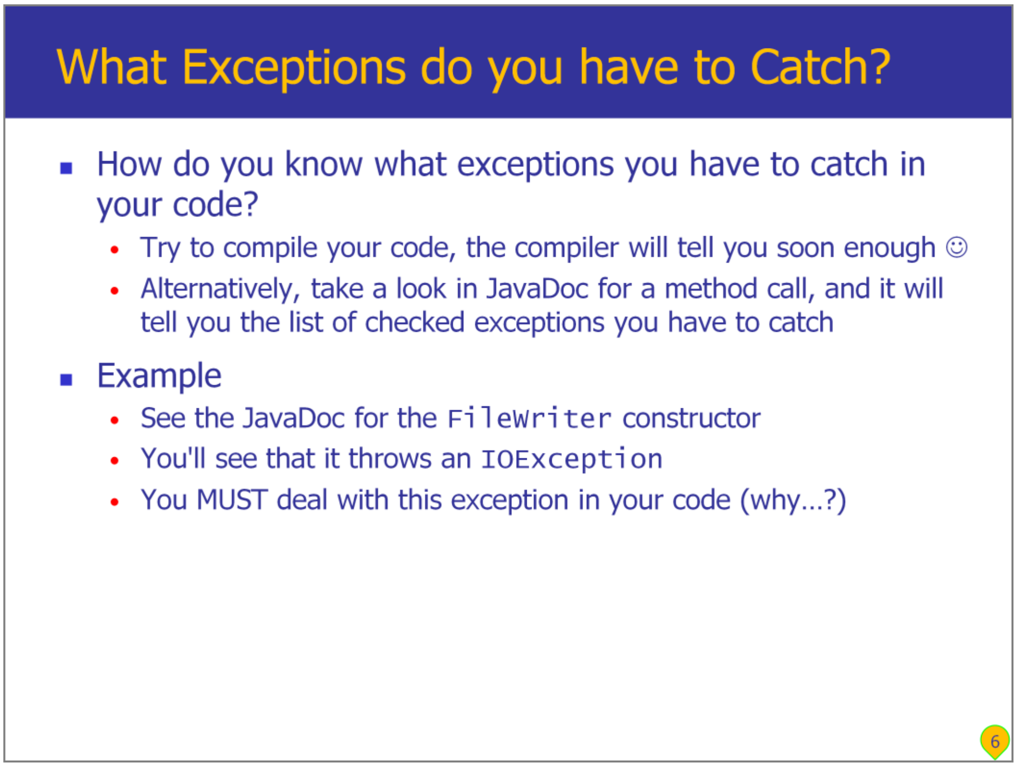
Categories of Exceptions in Java

- There are lots of things that can go wrong in a Java app
  - Therefore, there are lots of different exception classes
  - Each exception class represents a different kind of problem
  - The Java library defines hundreds of standard exception classes (see later for some examples)
- Java classifies exceptions into 3 categories:

Throwable
Error
Exception
System error classes
You SHOULDN'T catch these, they're too serious!
RuntimeException
Checked exception classes
You MUST catch these, or you'll get a compiler error.
Unchecked exception classes
You CAN catch these, but prevention is better.
5

Java defines a large number of standard exception classes, and you can also define your own custom exception classes relevant to your particular business domain. The exception class hierarchy in Java is more complicated than you might imagine:

- All exception/error classes ultimately inherit from `Throwable`, which obviously inherits from `Object` further up the inheritance tree.
- There's a distinction between errors and exceptions:
  - The `Error` class, and its subclasses, represent serious internal errors that are too dire for you to handle in your own code.
  - The `Exception` class, and its subclasses, represent exceptional situations that are less serious and can be handled in your code.
- Underneath the `Exception` class, there are two distinct kinds of exception, i.e. runtime exceptions and checked exceptions:
  - A runtime exception is a class that inherits from `RuntimeException` and represents an exception thrown by the JVM runtime (such as a null object reference or an arithmetic overflow). You can catch these exceptions if you like, but you don't have to, and generally prevention would be a better recourse.
  - A checked exception doesn't inherit from `RuntimeException`, i.e. it inherits directly from `Exception`. You must handle these exceptions in your code, otherwise you'll get a compiler error. Hence the term "checked exception", because the compiler checks that you handle it.

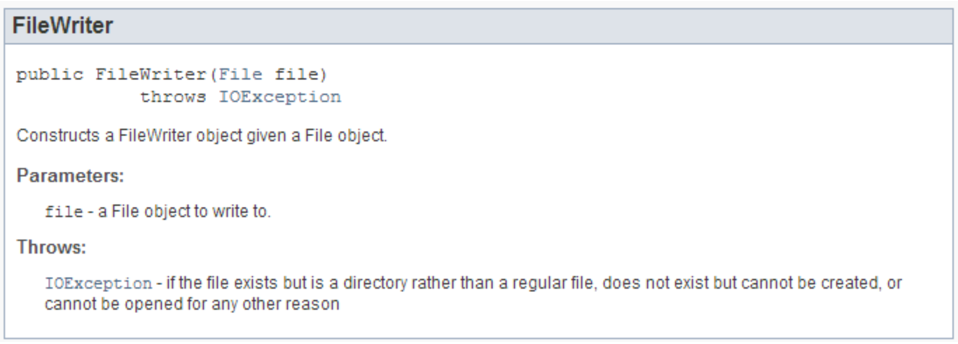


What Exceptions do you have to Catch?

- How do you know what exceptions you have to catch in your code?
  - Try to compile your code, the compiler will tell you soon enough ☺
  - Alternatively, take a look in JavaDoc for a method call, and it will tell you the list of checked exceptions you have to catch
- Example
  - See the JavaDoc for the `FileWriter` constructor
  - You'll see that it throws an `IOException`
  - You MUST deal with this exception in your code (why...?)

6

Here's the JavaDoc documentation for the `FileWriter` constructor.



FileWriter

```
public FileWriter(File file)
           throws IOException
```

Constructs a FileWriter object given a File object.

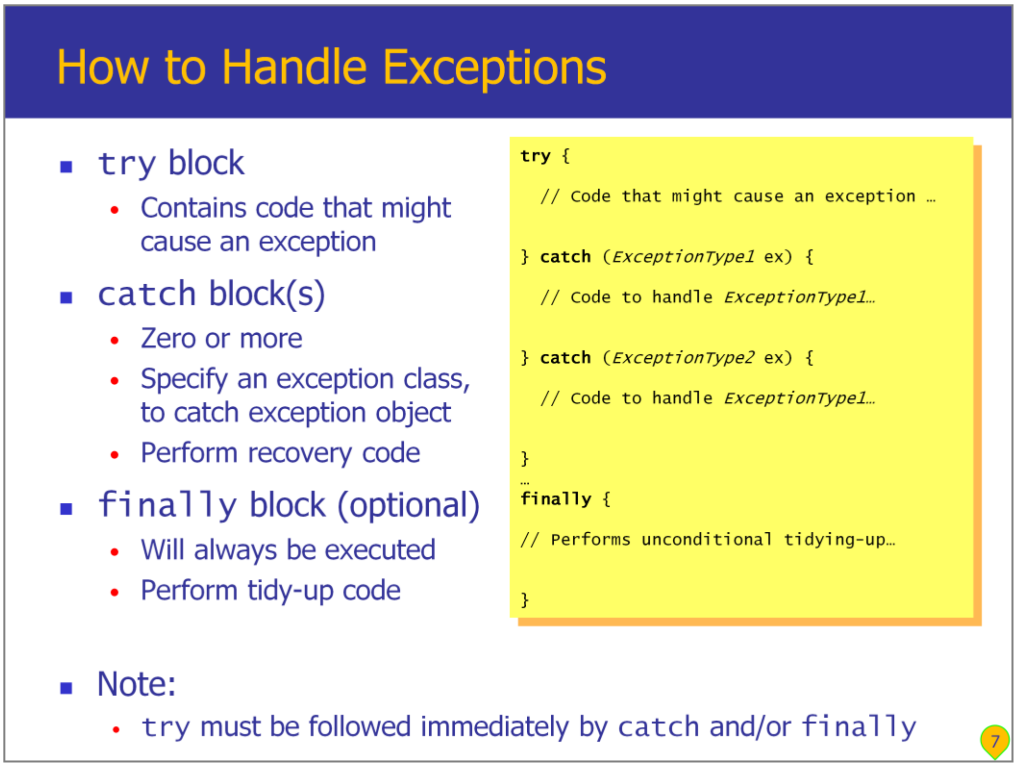
Parameters:

`file` - a File object to write to.

Throws:

`IOException` - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

Note the `throws` clause at the end of the method signature. This tells you what types of checked exceptions might occur when you invoke this method. In other words, your calling code must be prepared to catch these exceptions. In this particular example, your calling code must be prepared to handle an `IOException` exception.



How to Handle Exceptions

- `try` block
  - Contains code that might cause an exception
- `catch` block(s)
  - Zero or more
  - Specify an exception class, to catch exception object
  - Perform recovery code
- `finally` block (optional)
  - Will always be executed
  - Perform tidy-up code

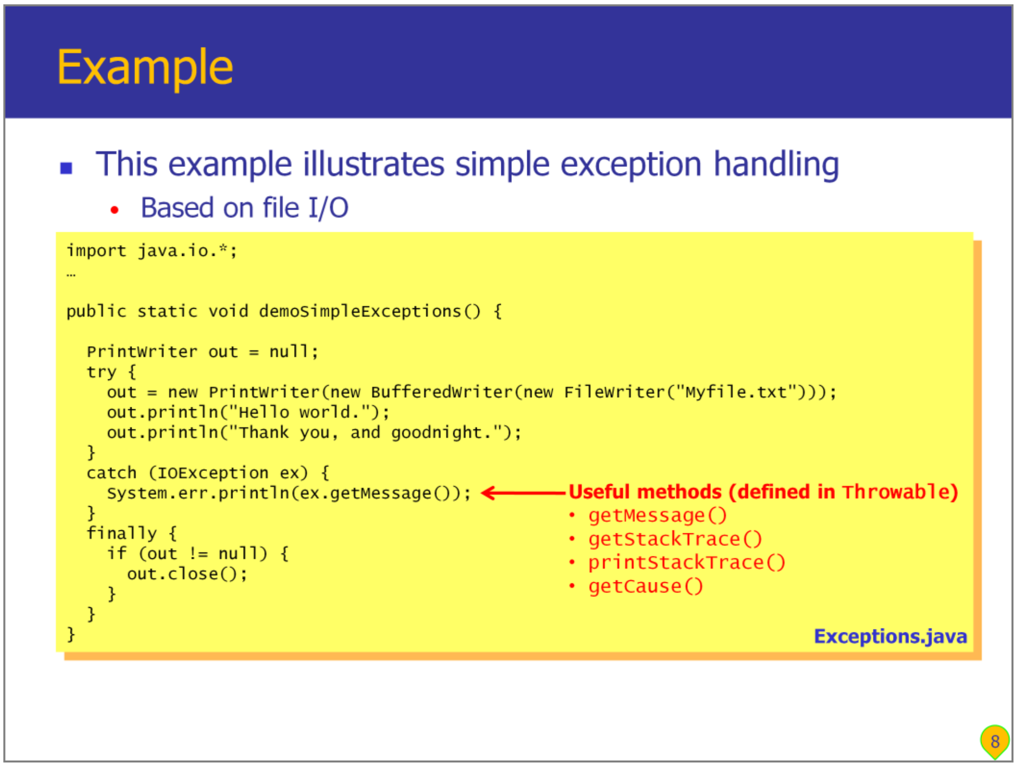
```
try {
  // Code that might cause an exception …

} catch (ExceptionType1 ex) {
  // Code to handle ExceptionType1…

} catch (ExceptionType2 ex) {
  // Code to handle ExceptionType1…

}
…
finally {
// Performs unconditional tidying-up…

}
```

- Note:
  - `try` must be followed immediately by `catch` and/or `finally`

7

This slide shows all the syntax for writing exception-safe code. Note the following points in particular:

- A `try` block might contain code that throws multiple different types of exception. If an exception occurs, the rest of the `try` block is skipped and control passes immediately to the appropriate `catch` handler block beneath the `try` block.
- Optionally, you can append a `finally` block at the end. The `finally` block will always be executed, regardless of what happened in the `try/catch` blocks.



Example

- This example illustrates simple exception handling
  - Based on file I/O

```
import java.io.*;
…

public static void demoSimpleExceptions() {

  PrintWriter out = null;
  try {
    out = new PrintWriter(new BufferedWriter(new FileWriter("Myfile.txt")));
    out.println("Hello world.");
    out.println("Thank you, and goodnight.");
  }
  catch (IOException ex) {
    System.err.println(ex.getMessage());
  }
  finally {
    if (out != null) {
      out.close();
    }
  }
}
```

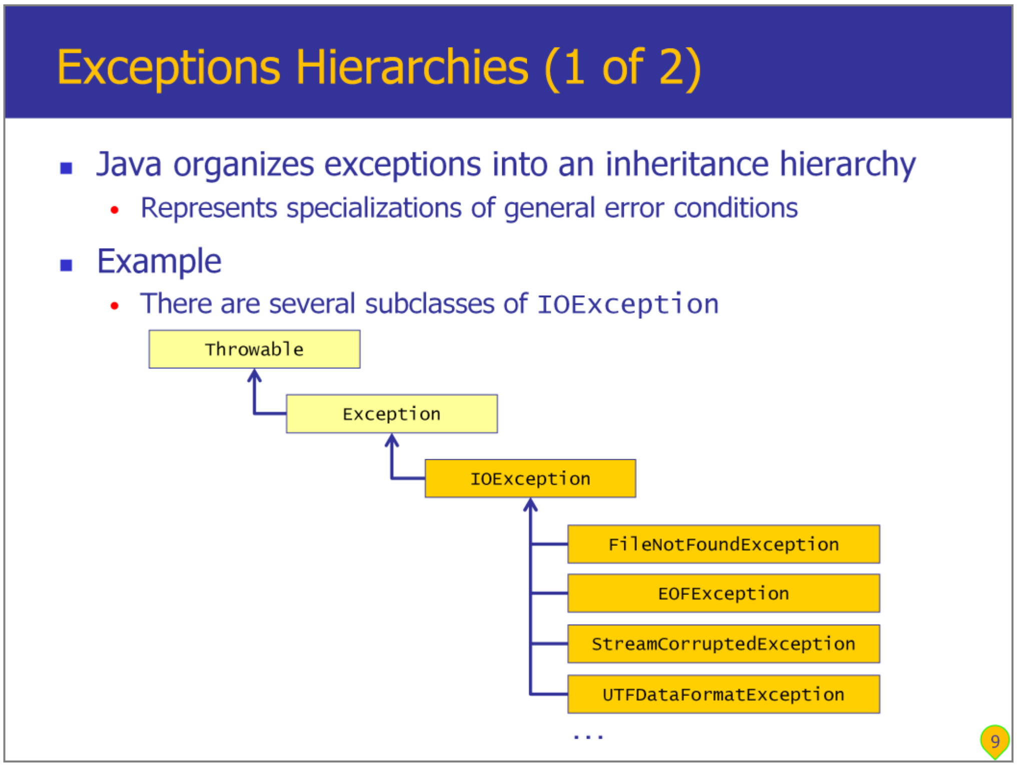
Useful methods (defined in `Throwable`)
- `getMessage()`
- `getStackTrace()`
- `printStackTrace()`
- `getCause()`

Exceptions.java

8

This slide shows an example of how to handle `IOException` exceptions, which might arise when you perform any file-handling operation.

Note that all exception classes have a `getMessage()` method that describes the problem, plus various other methods that give additional information about the exception (such as the stack trace, for diagnostics and logging purposes).

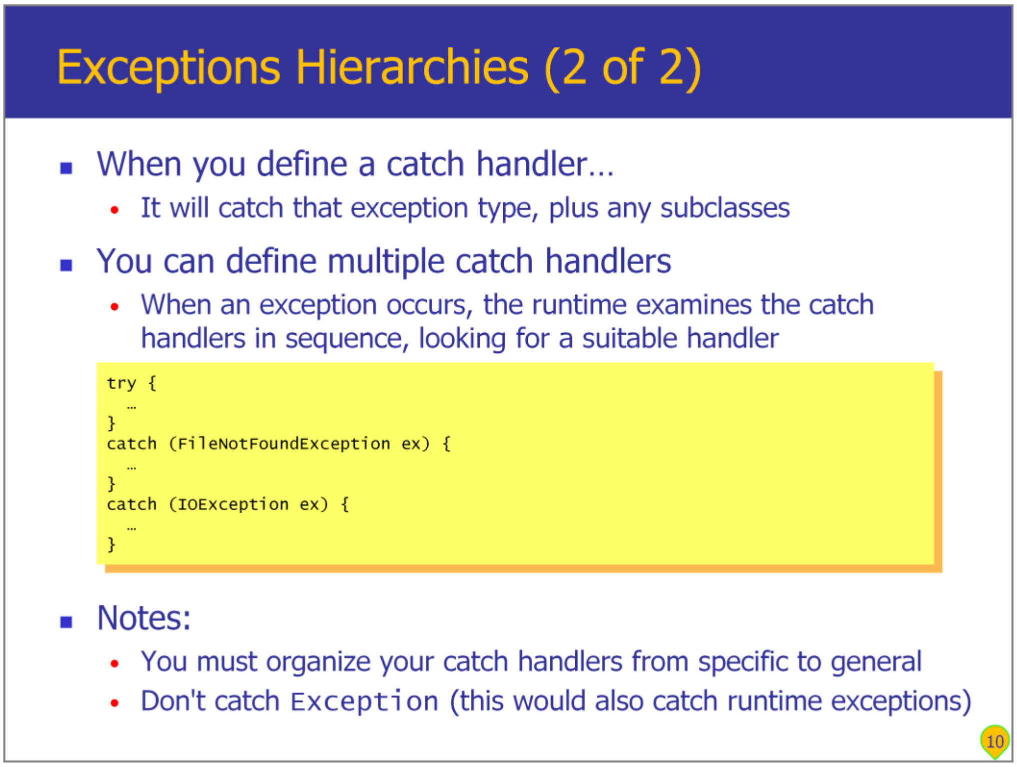


Exceptions Hierarchies (1 of 2)

- Java organizes exceptions into an inheritance hierarchy
  - Represents specializations of general error conditions
- Example
  - There are several subclasses of `IOException`

Throwable
Exception
IOException
FileNotFoundException
EOFException
StreamCorruptedException
UTFDataFormatException
...
9

Java defines a large number of exception classes, arranged in a meaningful inheritance hierarchy. The diagram on the slide shows the `IOException` class, plus some of its subclasses that represent specific kinds of I/O exception.



Exceptions Hierarchies (2 of 2)

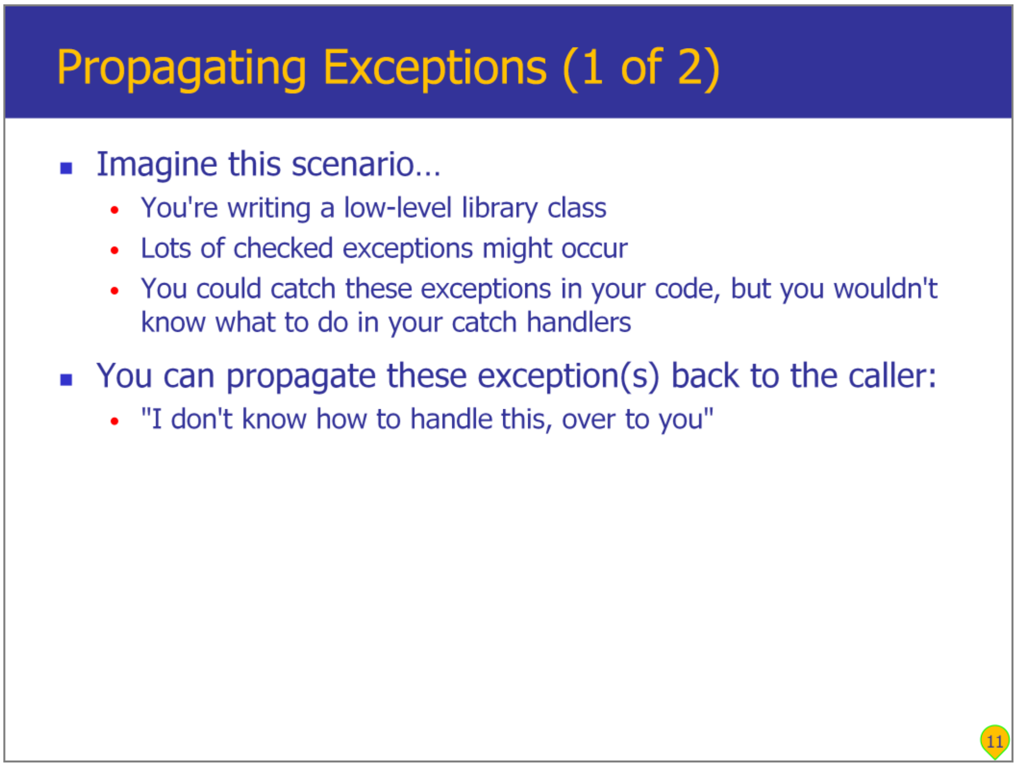
- When you define a catch handler…
  - It will catch that exception type, plus any subclasses
- You can define multiple catch handlers
  - When an exception occurs, the runtime examines the catch handlers in sequence, looking for a suitable handler

```
try {
  …
}
catch (FileNotFoundException ex) {
  …
}
catch (IOException ex) {
  …
}
```

- Notes:
  - You must organize your catch handlers from specific to general
  - Don't catch `Exception` (this would also catch runtime exceptions)

10

You must organize your `catch` blocks in specific-to-general order. For example, if you want to catch `FileNotFoundException` and `IOException` exceptions, you must put the `FileNotFoundException` handler before the `IOException` handler.



Propagating Exceptions (1 of 2)

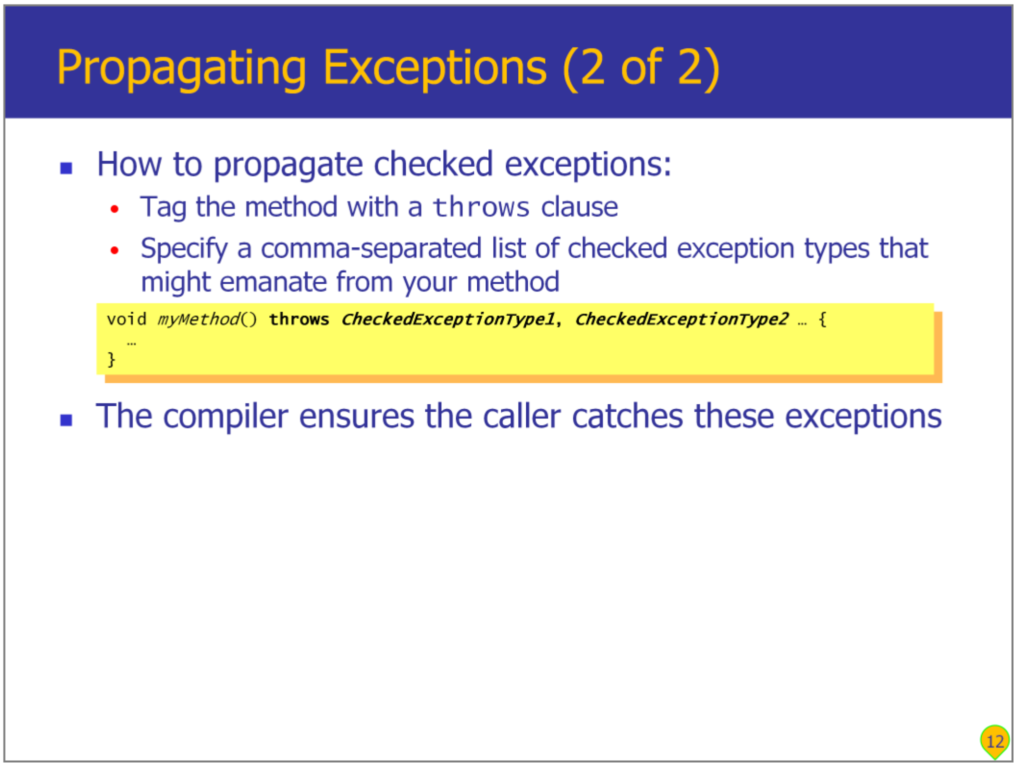
- Imagine this scenario…
  - You're writing a low-level library class
  - Lots of checked exceptions might occur
  - You could catch these exceptions in your code, but you wouldn't know what to do in your catch handlers
- You can propagate these exception(s) back to the caller:
  - "I don't know how to handle this, over to you"

11

Imagine a scenario where you're writing some low-level business logic. You call a database API that throws an exception. So you duly catch the exception and do some local tidying up (e.g. flush some memory, remove some items from collections, delete some files, etc.).

Imagine you also want to display an error message to the user. The problem is, your low-level business logic doesn't contain any UI capabilities; it might not even know what kind of UI the application is using (e.g. is it a web app, a desktop app, a mobile app, etc.).

In such situations, you can allow the exception to propagate outwards to the code that called you. The calling code can then do some additional higher-level exception processing, such as displaying an error message to the user.



Propagating Exceptions (2 of 2)

- How to propagate checked exceptions:
  - Tag the method with a `throws` clause
  - Specify a comma-separated list of checked exception types that might emanate from your method

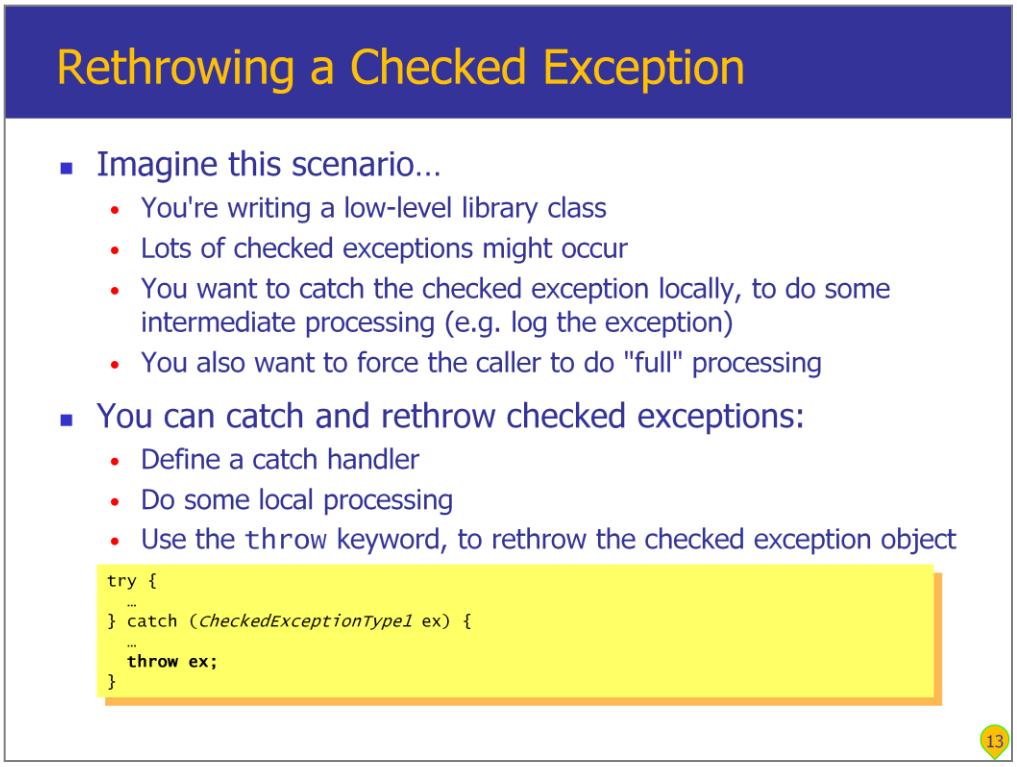
```
void myMethod() throws CheckedExceptionType1, CheckedExceptionType2 … {
  …
}
```

- The compiler ensures the caller catches these exceptions

12

To allow an exception to propagate out of your method, append a `throws` clause at the end of your method signature.

The `throws` clause specifies a comma-separated list of checked exceptions that might emanate from your method. The compiler doesn't then force your method to handle the exceptions; instead, this responsibility falls back on the code that called you. It's a bit like having sloping shoulders ☺.



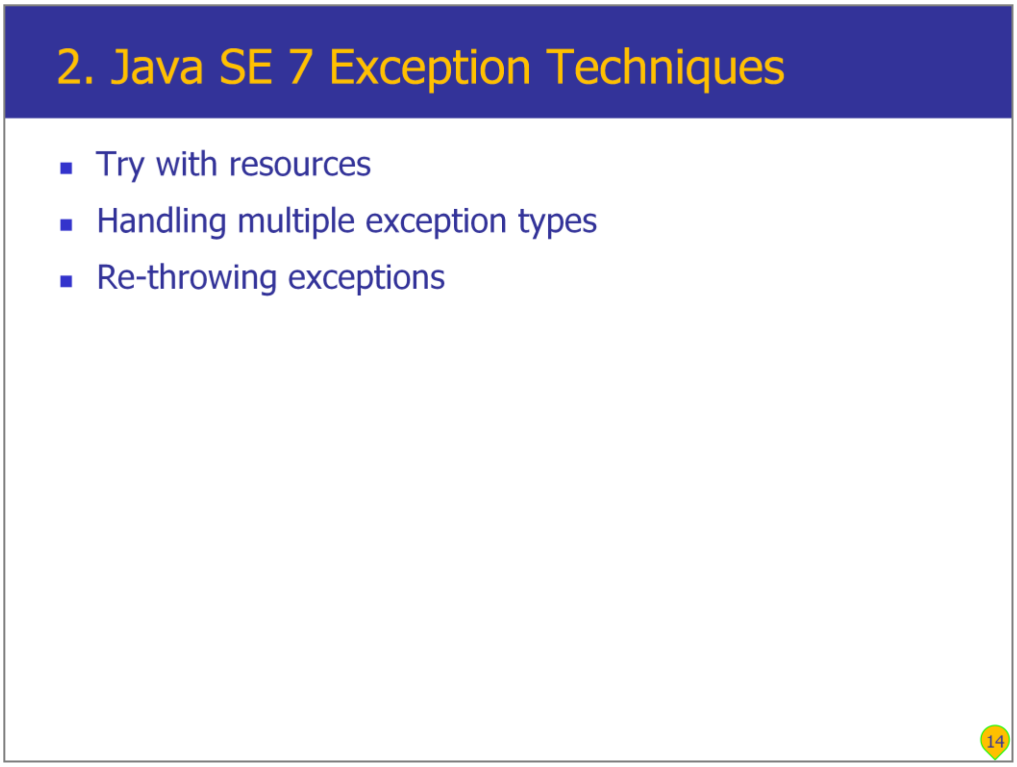
Rethrowing a Checked Exception

- Imagine this scenario…
  - You're writing a low-level library class
  - Lots of checked exceptions might occur
  - You want to catch the checked exception locally, to do some intermediate processing (e.g. log the exception)
  - You also want to force the caller to do "full" processing
- You can catch and rethrow checked exceptions:
  - Define a catch handler
  - Do some local processing
  - Use the `throw` keyword, to rethrow the checked exception object

```
try {
  …
} catch (CheckedExceptionType1 ex) {
  …
  throw ex;
}
```

13

This slide shows how to catch an exception, do some localized processing, and then re-throw the same exception so that it can be handled in a more sophisticated manner by the calling code.

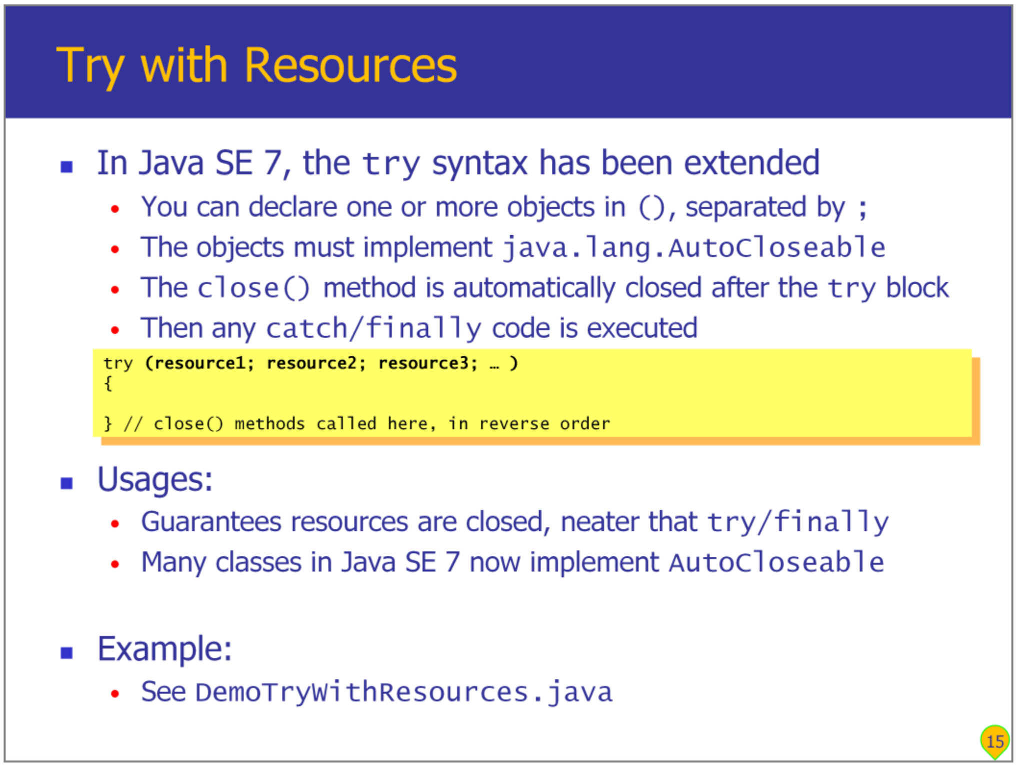


2. Java SE 7 Exception Techniques

- Try with resources
- Handling multiple exception types
- Re-throwing exceptions

14

In this section, we'll take a look at three new exception-related language features that were introduced in the Java language in Java SE version 7. These features make it easier to write tight and robust exception-safe code.



Try with Resources

- In Java SE 7, the `try` syntax has been extended
  - You can declare one or more objects in `()`, separated by `;`
  - The objects must implement `java.lang.AutoCloseable`
  - The `close()` method is automatically closed after the `try` block
  - Then any `catch/finally` code is executed

```
try (resource1; resource2; resource3; … )
{

} // close() methods called here, in reverse order
```

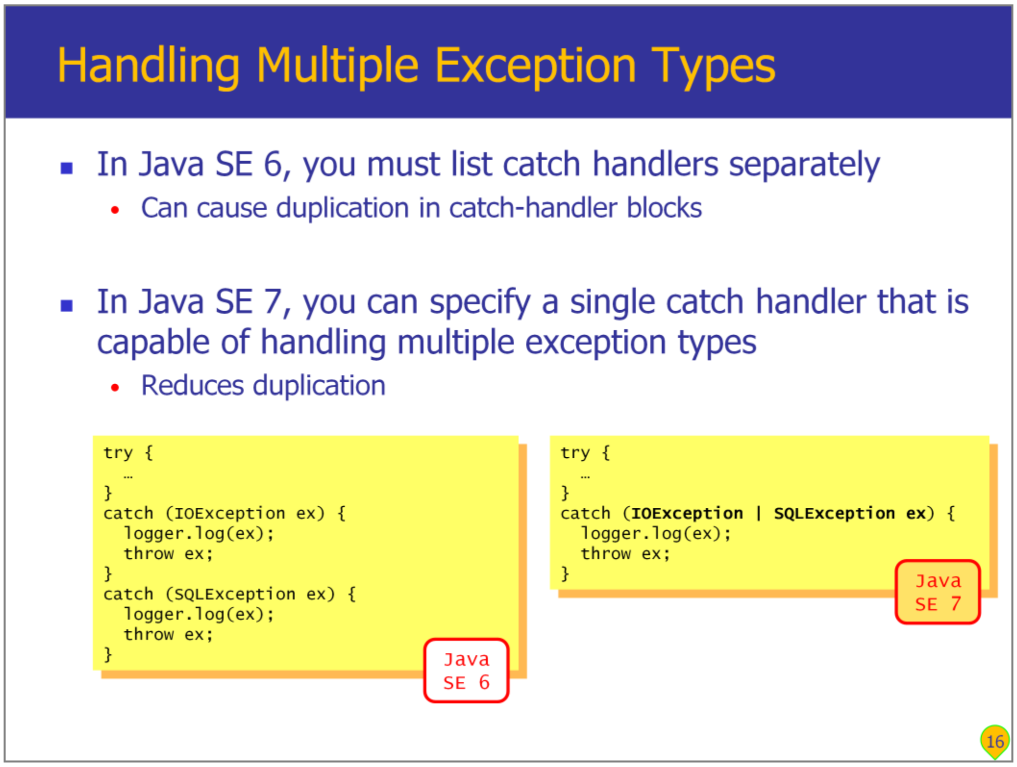
- Usages:
  - Guarantees resources are closed, neater that `try/finally`
  - Many classes in Java SE 7 now implement `AutoCloseable`
- Example:
  - See `DemoTryWithResources.java`

15

Many resource-related classes (e.g. file-related classes) have been refactored in Java SE 7 so that they implement a new interface named `AutoCloseable`. Any such type can be used in a try-with-resources statement as shown in the slide.

The point is that the `close()` method is automatically invoked when the auto-closeable object(s) go out of scope at the end of the `try` block. The `close()` method in each auto-closeable class performs the necessary tasks to release the resource (e.g. close the file).

The demo project provides several specific examples.



Handling Multiple Exception Types

- In Java SE 6, you must list catch handlers separately
  - Can cause duplication in catch-handler blocks
- In Java SE 7, you can specify a single catch handler that is capable of handling multiple exception types
  - Reduces duplication

```
try {
  …
}
catch (IOException ex) {
  logger.log(ex);
  throw ex;
}
catch (SQLException ex) {
  logger.log(ex);
  throw ex;
}
```

Java SE 6

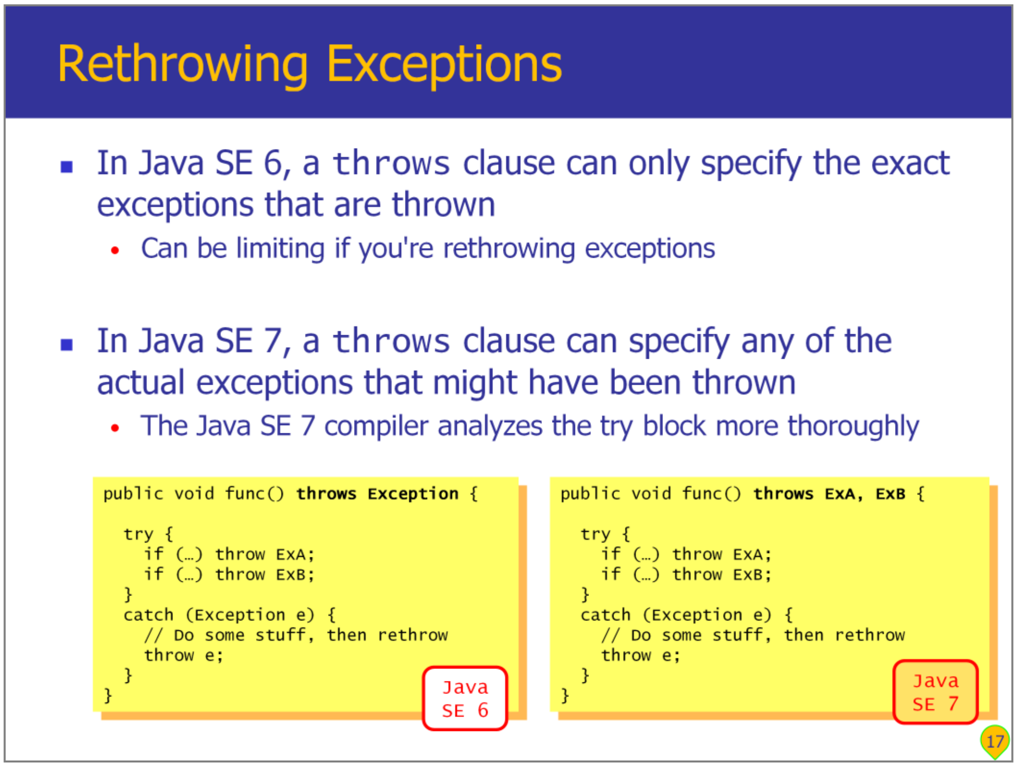
```
try {
  …
}
catch (IOException | SQLException ex) {
  logger.log(ex);
  throw ex;
}
```

Java SE 7

16

Consider the two code examples in the slide:

- The code example on the left is implemented in Java SE 6. The code catches two different types of exception - `IOException` and `SQLException` - and happens to perform the same processing in each catch handler.
- The code example on the right is implemented in Java SE 7, where you can specify multiple exception types in the same catch handler (separated by a | symbol). This reduces duplication between catch handlers.



Rethrowing Exceptions

- In Java SE 6, a `throws` clause can only specify the exact exceptions that are thrown
  - Can be limiting if you're rethrowing exceptions
- In Java SE 7, a `throws` clause can specify any of the actual exceptions that might have been thrown
  - The Java SE 7 compiler analyzes the try block more thoroughly

```
public void func() throws Exception {

  try {
    if (…) throw ExA;
    if (…) throw ExB;
  }
  catch (Exception e) {
    // Do some stuff, then rethrow
    throw e;
  }
}
```

Java SE 6

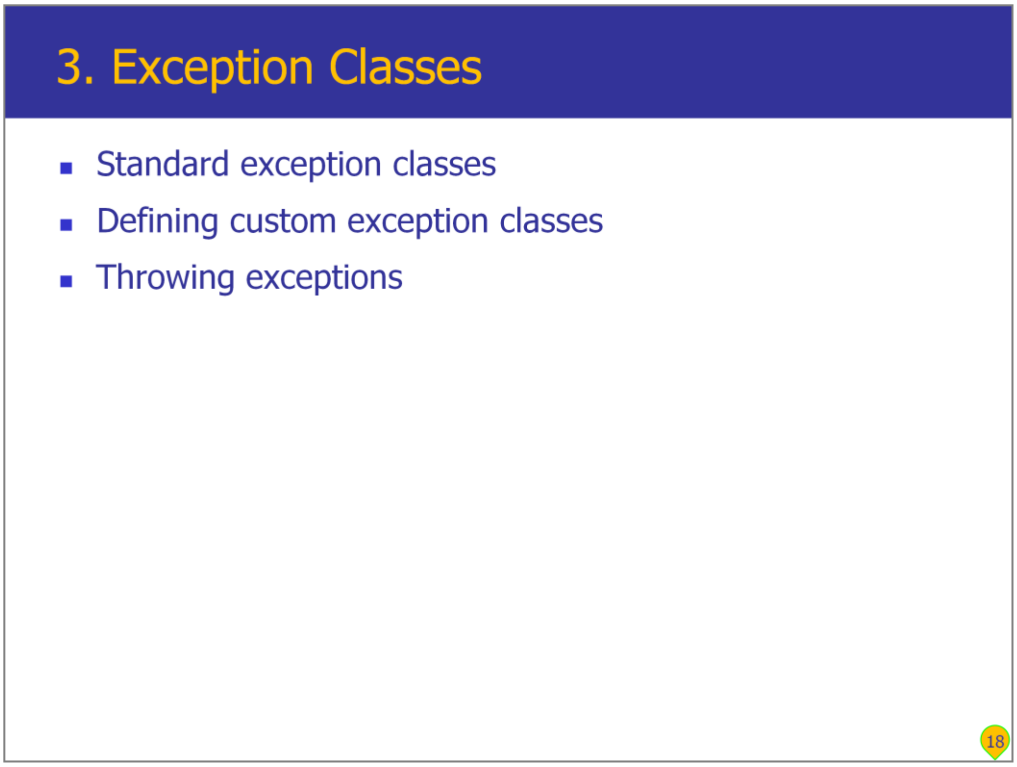
```
public void func() throws ExA, ExB {

  try {
    if (…) throw ExA;
    if (…) throw ExB;
  }
  catch (Exception e) {
    // Do some stuff, then rethrow
    throw e;
  }
}
```

Java SE 7

17

Consider the two code examples in the slide:

- The code example on the left is implemented in Java SE 6. The code defines a general-purpose catch handler that can catch any kind of `Exception` (this is probably a bit too loose to be honest - you shouldn't really be catching all exceptions like this). Anyway, the catch handler is certainly capable of catching `ExA` and `ExB` exceptions. The idea is that the catch handler will do some local tidying up, and then re-throw the exception so that a higher-level function can do some more significant processing. The problem is that in Java SE 6, the `throws` clause in the method signature can only specify the type of exception specified in the catch handler - i.e. `Exception`. This means client code is obliged to also catch `Exception` (even though it really only ought to worry about `ExA` and `ExB`).
- Java SE 7 relaxes the rules for the `throws` clause, so that you can now specify precisely the types of exception that can actually occur in the `try` block (i.e. `ExA` and `ExB` in this example), rather than being constrained to specify the type of exception in the `catch` block. Client code now knows precisely what types of exceptions it needs to catch.

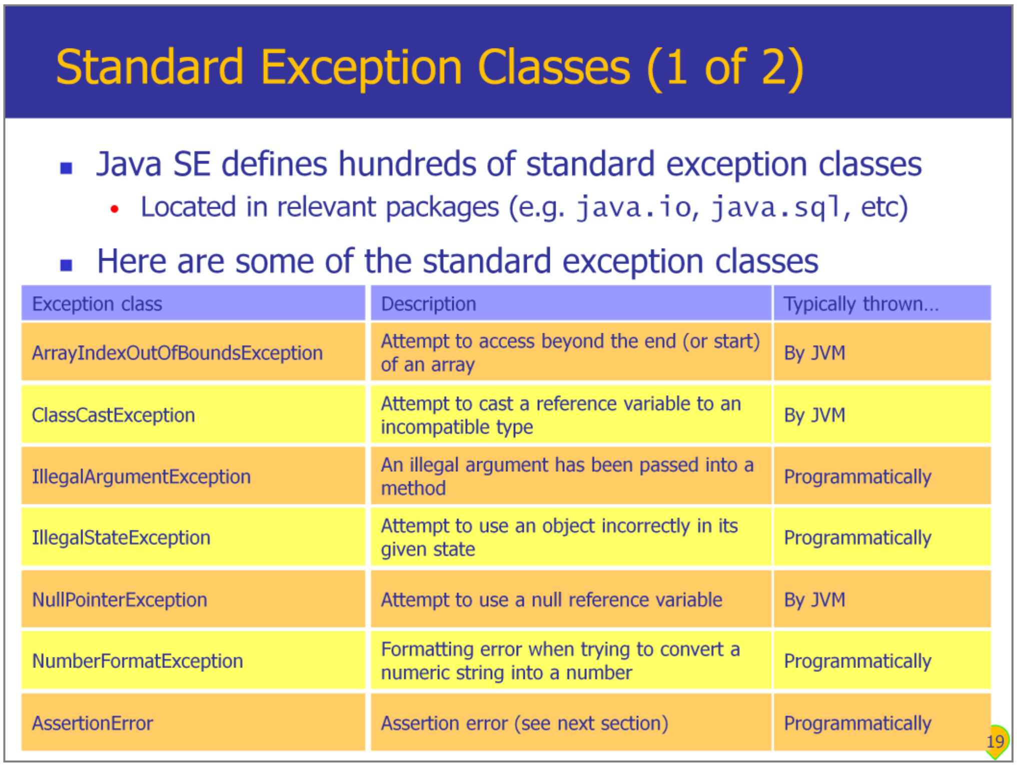


3. Exception Classes

- Standard exception classes
- Defining custom exception classes
- Throwing exceptions

18

The Java SE library defines a very large number of standard exception classes. In this section, we'll discuss some of the more common ones, and provide some recommendations on how to handle them. We'll also describe how to throw exceptions, and how to define custom exception classes relevant to your particular domain.

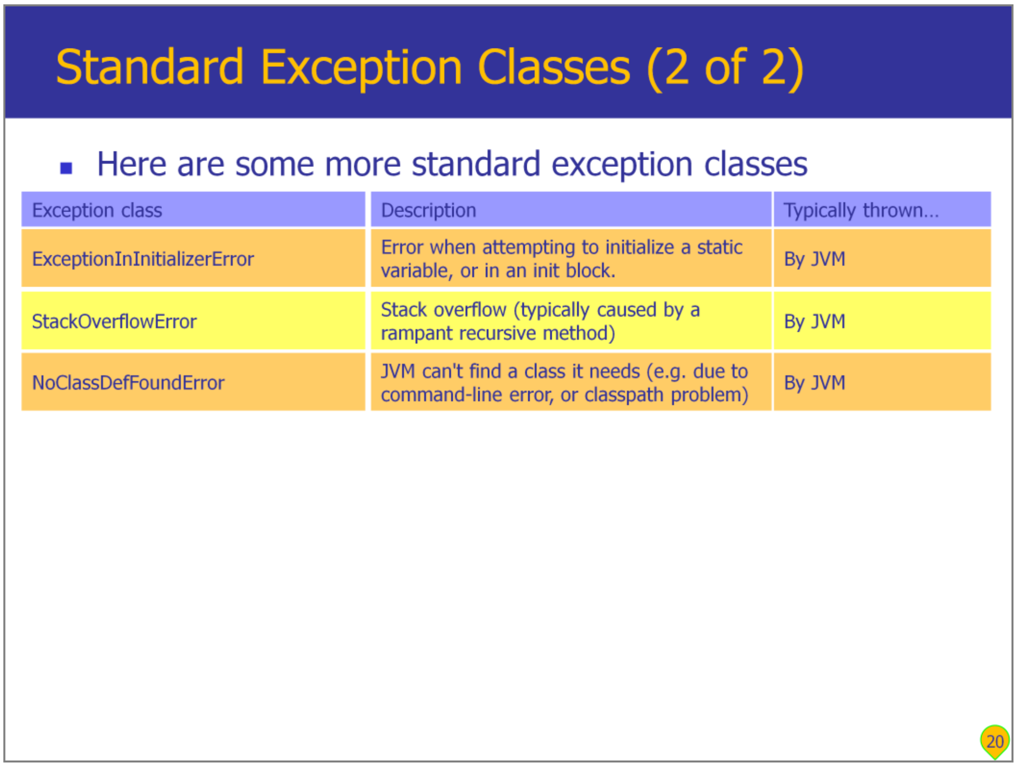


Standard Exception Classes (1 of 2)

- Java SE defines hundreds of standard exception classes
  - Located in relevant packages (e.g. `java.io`, `java.sql`, etc)
- Here are some of the standard exception classes

Exception class	Description	Typically thrown…
ArrayIndexOutOfBoundsException	Attempt to access beyond the end (or start) of an array	By JVM
ClassCastException	Attempt to cast a reference variable to an incompatible type	By JVM
IllegalArgumentException	An illegal argument has been passed into a method	Programmatically
IllegalStateException	Attempt to use an object incorrectly in its given state	Programmatically
NullPointerException	Attempt to use a null reference variable	By JVM
NumberFormatException	Formatting error when trying to convert a numeric string into a number	Programmatically
AssertionError	Assertion error (see next section)	Programmatically

Here are some of the commonly occurring exception classes in the Java SE library, along with a description of why they occur. It's good to be aware of the kinds of things that can go wrong, so you can pre-empt then if possible.



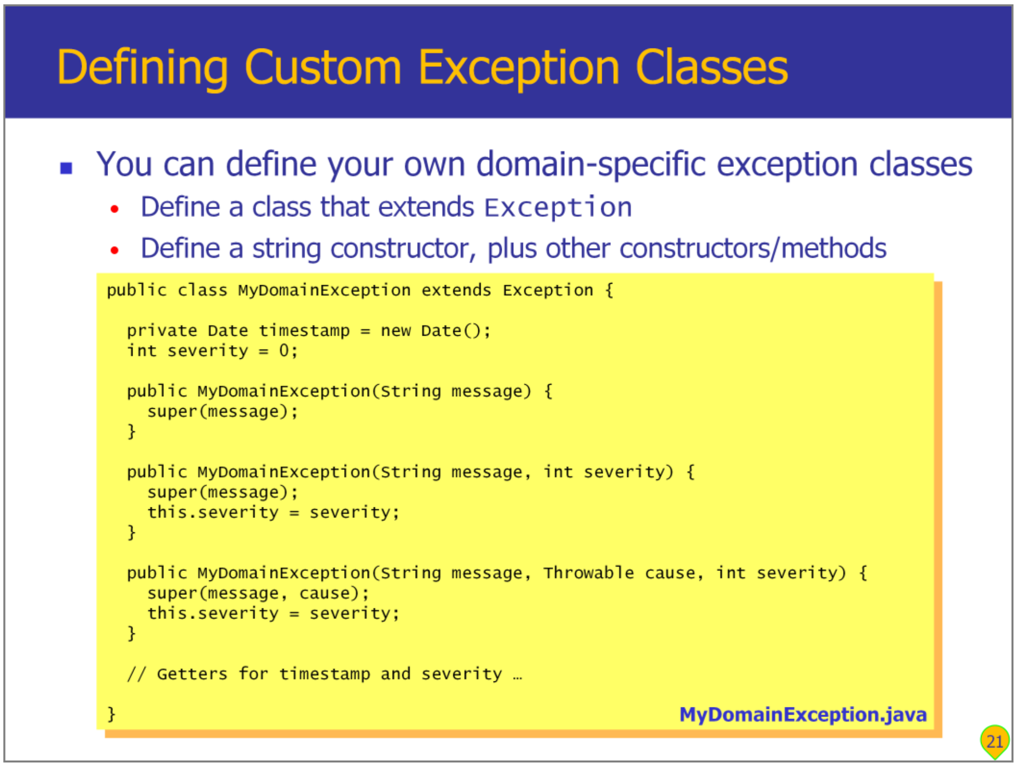
Standard Exception Classes (2 of 2)

- Here are some more standard exception classes

Exception class	Description	Typically thrown…
ExceptionInInitializerError	Error when attempting to initialize a static variable, or in an init block.	By JVM
StackOverflowError	Stack overflow (typically caused by a rampant recursive method)	By JVM
NoClassDefFoundError	JVM can't find a class it needs (e.g. due to command-line error, or classpath problem)	By JVM

20

This slide continues the list of common exception classes.



Defining Custom Exception Classes

- You can define your own domain-specific exception classes
  - Define a class that extends `Exception`
  - Define a string constructor, plus other constructors/methods

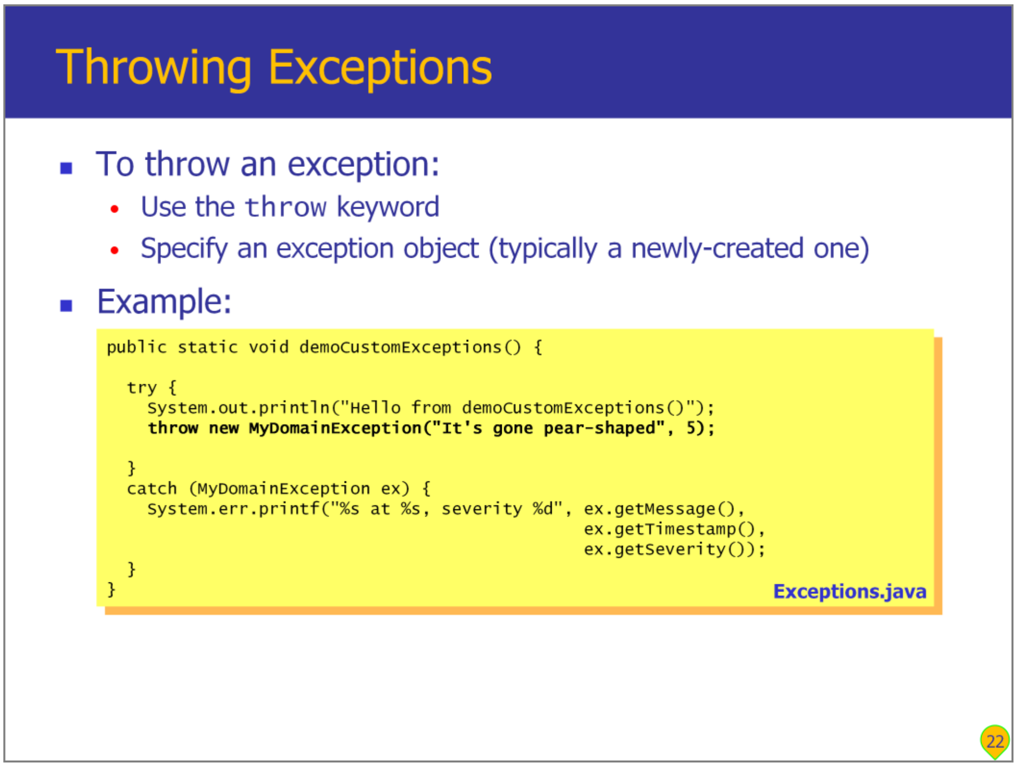
```
public class MyDomainException extends Exception {

  private Date timestamp = new Date();
  int severity = 0;

  public MyDomainException(String message) {
    super(message);
  }

  public MyDomainException(String message, int severity) {
    super(message);
    this.severity = severity;
  }

  public MyDomainException(String message, Throwable cause, int severity) {
    super(message, cause);
    this.severity = severity;
  }

  // Getters for timestamp and severity …

}
```

MyDomainException.java

21

You are advised to define custom exception classes to represent business-related exception conditions in your code.

The example in the slide is a template for defining custom exception classes. Basically it's a matter of providing an appropriate set of constructors, plus any additional domain-specific fields that help to give some context about the exception.



Throwing Exceptions

- To throw an exception:
  - Use the `throw` keyword
  - Specify an exception object (typically a newly-created one)
- Example:

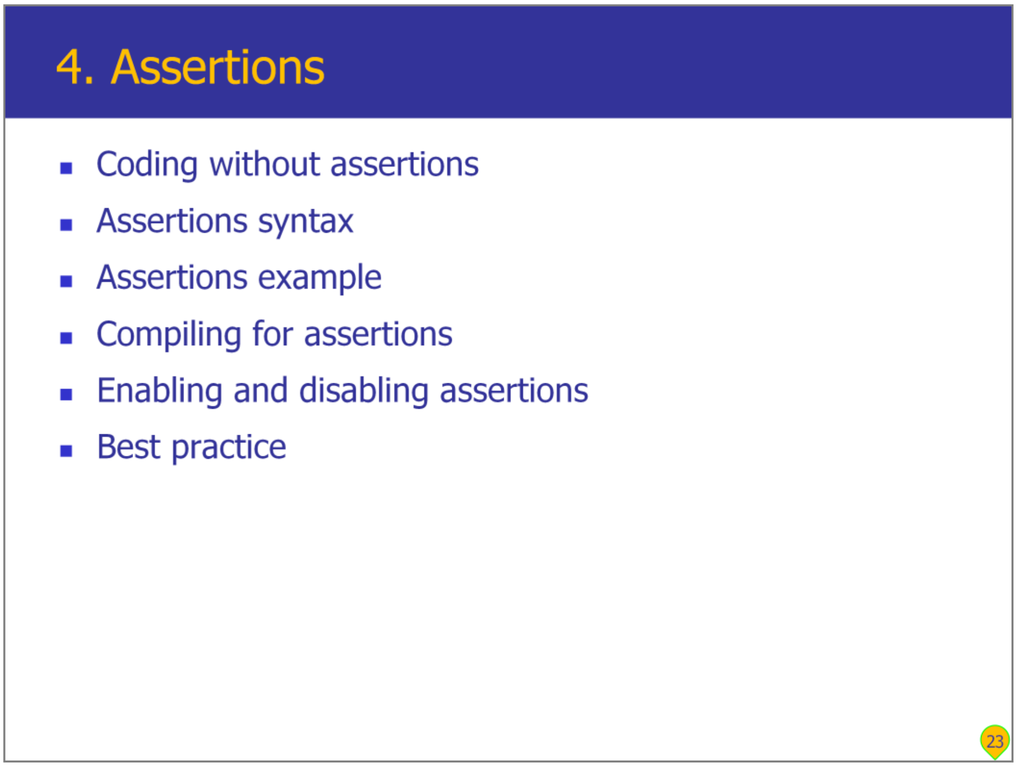
```
public static void demoCustomExceptions() {

  try {
    System.out.println("Hello from demoCustomExceptions()");
    throw new MyDomainException("It's gone pear-shaped", 5);

  }
  catch (MyDomainException ex) {
    System.err.printf("%s at %s, severity %d", ex.getMessage(),
                                                ex.getTimestamp(),
                                                ex.getSeverity());
  }
}
```

Exceptions.java

22

To throw an exception, use the `throw` keyword followed by the exception object you want to throw.

The example in the slide is slightly artificial. It deliberately throws an exception and then catches it immediately in the same method. In reality, you'd just throw an exception and leave the calling method handle it.



4. Assertions

- Coding without assertions
- Assertions syntax
- Assertions example
- Compiling for assertions
- Enabling and disabling assertions
- Best practice

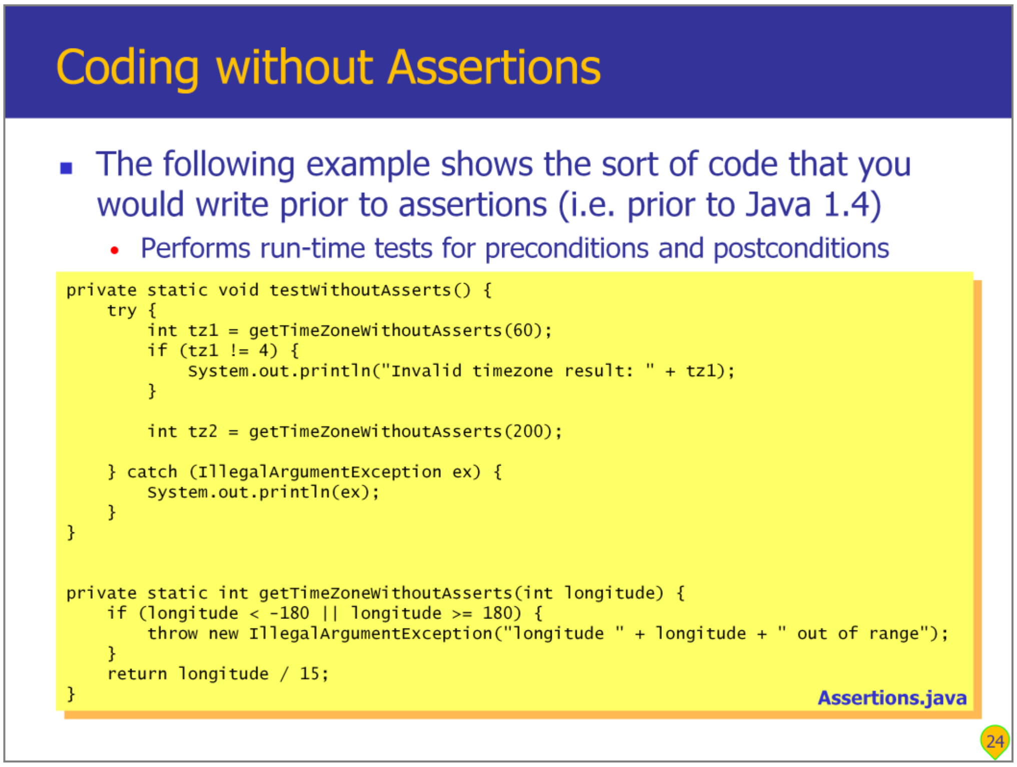
23

Assertions are boolean tests that verify whether specified conditions are true. This is how you use assertions in Java:

- Enable assertions during development
- Sprinkle `assert` statements in your code
- The `assert` statements highlight invalid values
- Disable assertions in the deployed code

Common uses for `assert` statements include:

- Preconditions
- Postconditions
- Class invariants



Coding without Assertions

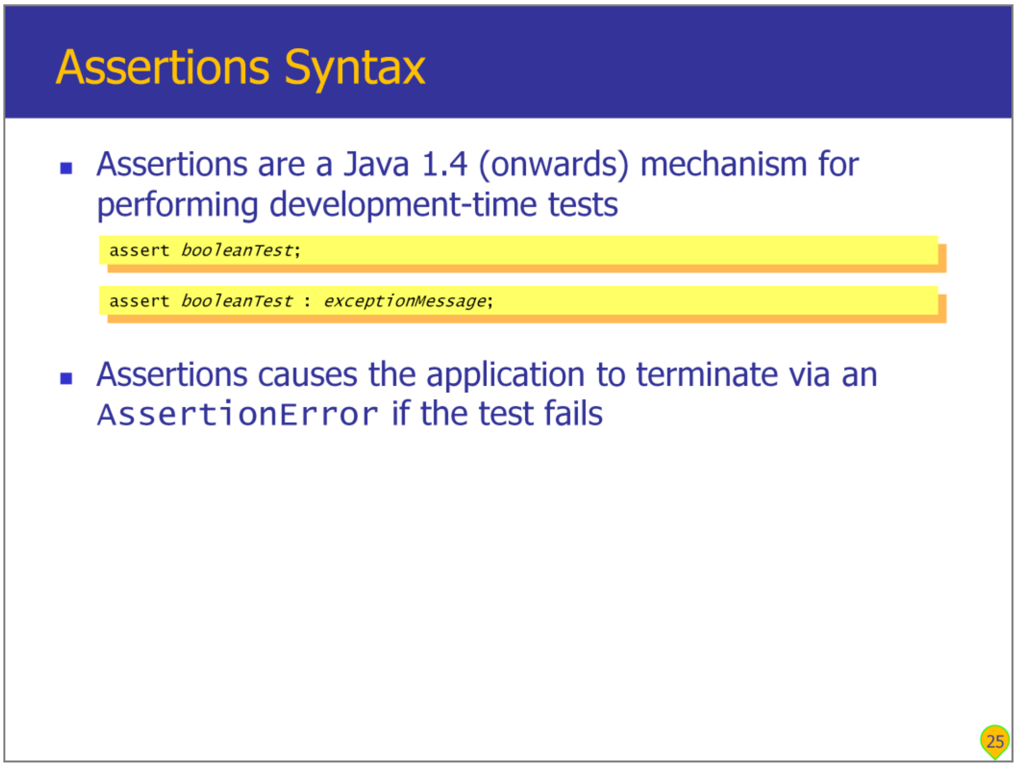
- The following example shows the sort of code that you would write prior to assertions (i.e. prior to Java 1.4)
  - Performs run-time tests for preconditions and postconditions

```
private static void testWithoutAsserts() {
    try {
        int tz1 = getTimeZoneWithoutAsserts(60);
        if (tz1 != 4) {
            System.out.println("Invalid timezone result: " + tz1);
        }

        int tz2 = getTimeZoneWithoutAsserts(200);

    } catch (IllegalArgumentException ex) {
        System.out.println(ex);
    }
}


private static int getTimeZoneWithoutAsserts(int longitude) {
    if (longitude < -180 || longitude >= 180) {
        throw new IllegalArgumentException("longitude " + longitude + " out of range");
    }
    return longitude / 15;
}
```

Assertions.java

24

Assertions are like mini if-tests, which you can disable when you're ready to deploy your code to production. Assertions were introduced in Java 1.4; before then, you would have to use full-blown if-tests to determine whether conditions were true.



Assertions Syntax

- Assertions are a Java 1.4 (onwards) mechanism for performing development-time tests

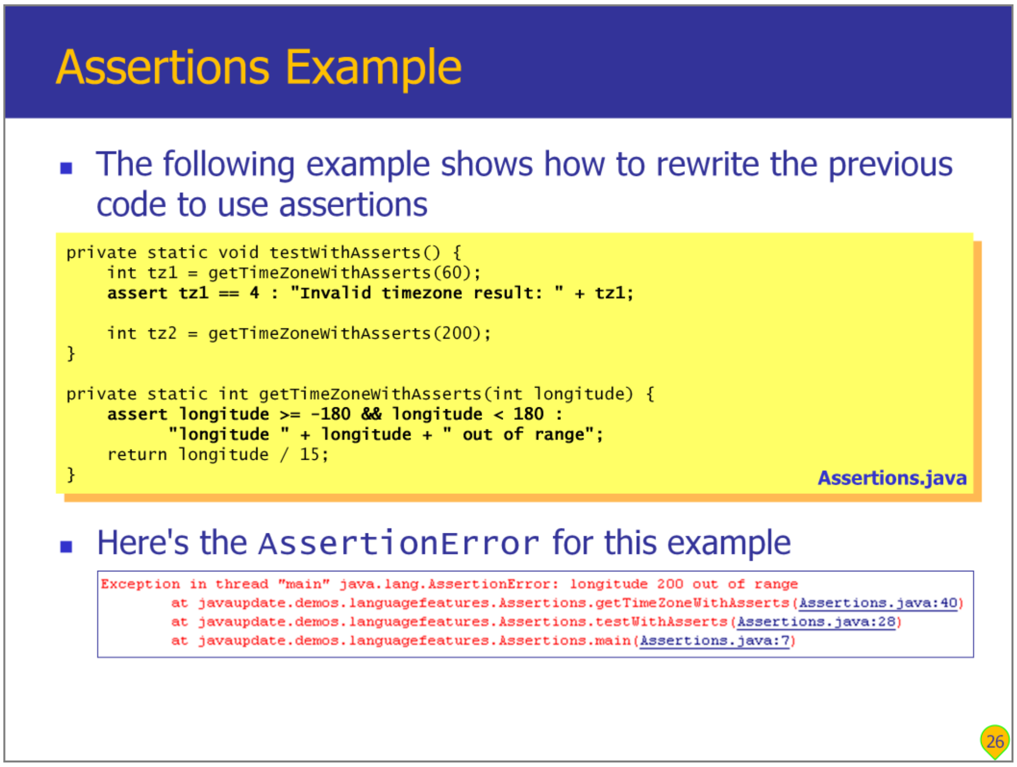
```
assert booleanTest;
```

```
assert booleanTest : exceptionMessage;
```

- Assertions causes the application to terminate via an `AssertionError` if the test fails

25

An `assert` statement includes a boolean test, followed by an optional message to be displayed if the test fails. Under the covers, a failed assertion causes an `AssertionError` to occur.



Assertions Example

- The following example shows how to rewrite the previous code to use assertions

```java
private static void testWithAsserts() {
    int tz1 = getTimeZoneWithAsserts(60);
    assert tz1 == 4 : "Invalid timezone result: " + tz1;

    int tz2 = getTimeZoneWithAsserts(200);
}

private static int getTimeZoneWithAsserts(int longitude) {
    assert longitude >= -180 && longitude < 180 :
          "longitude " + longitude + " out of range";
    return longitude / 15;
}
```

Assertions.java

- Here's the `AssertionError` for this example

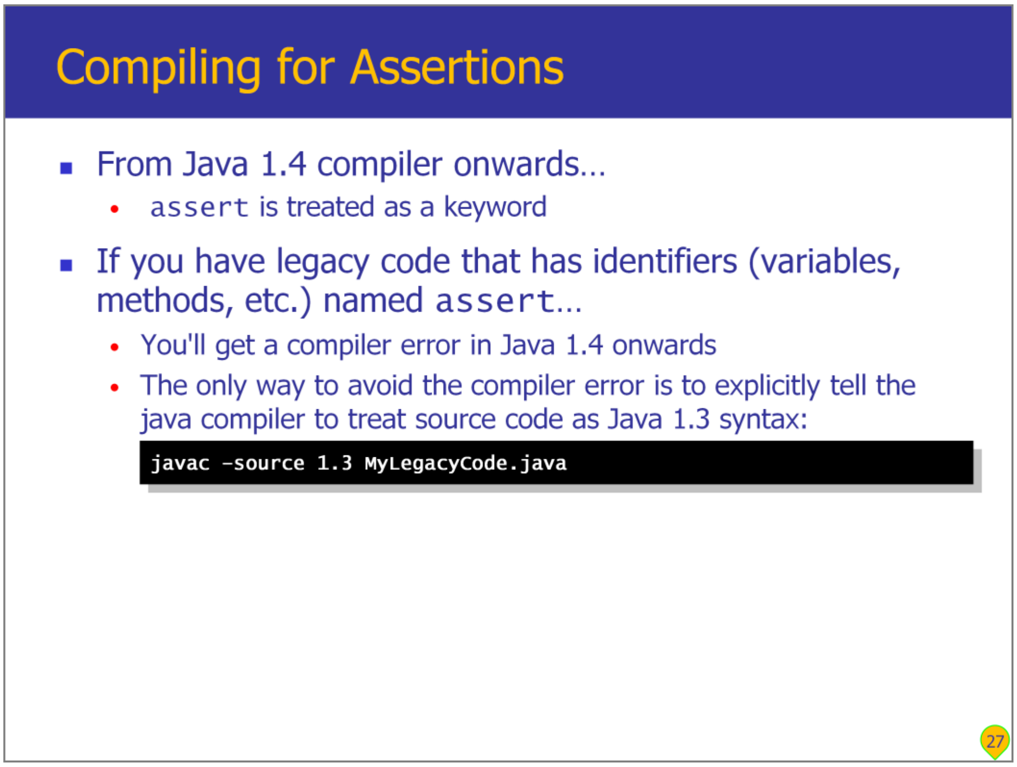
```
Exception in thread "main" java.lang.AssertionError: longitude 200 out of range
        at javaupdate.demos.languagefeatures.Assertions.getTimeZoneWithAsserts(Assertions.java:40)
        at javaupdate.demos.languagefeatures.Assertions.testWithAsserts(Assertions.java:28)
        at javaupdate.demos.languagefeatures.Assertions.main(Assertions.java:7)
```

26

This example shows how to perform assertions.

- The first method shows how to use assertions during testing. The method calls `getTimeZoneWithAsserts()` with the parameter 60. If `getTimeZoneWithAsserts()` is implemented correctly, we know it should return 4, so we write an `assert` statement to verify this is what happened. Effectively, we're testing the `getTimeZoneWithAsserts()` method here.
- The second method shows how to use assertions to define preconditions. The method asserts that the input value is in range.

The screenshot at the bottom of the slide shows what happens if an assertion fails. As you can see, it causes an `AssertionError` to occur.



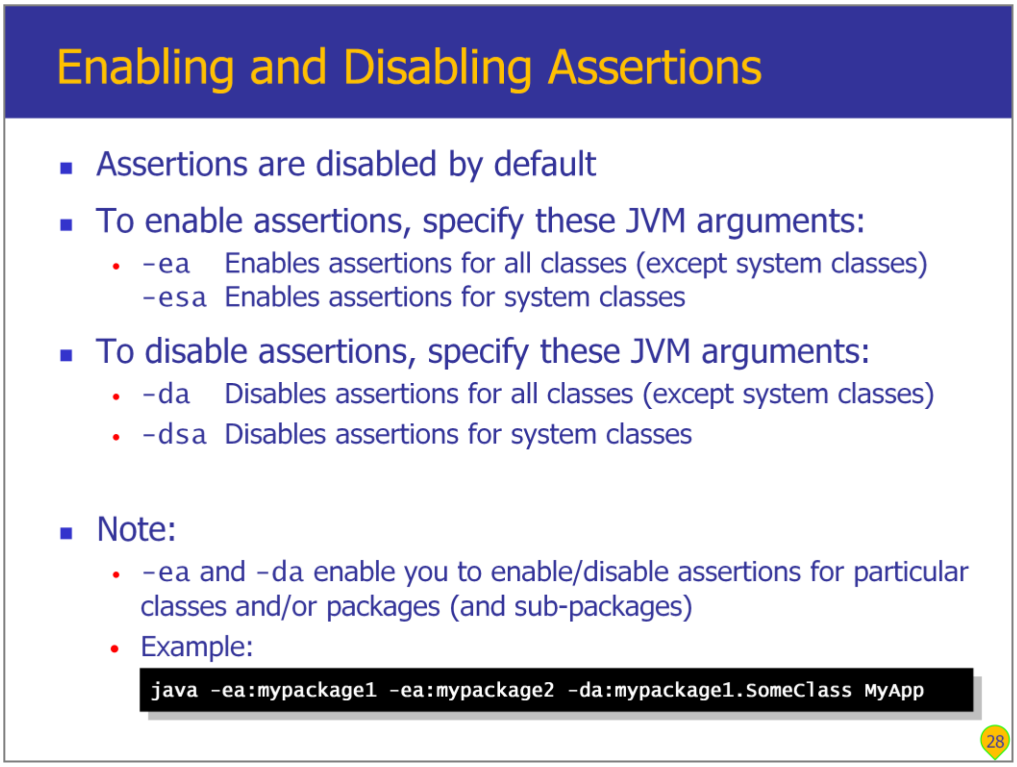
Compiling for Assertions

- From Java 1.4 compiler onwards…
  - `assert` is treated as a keyword
- If you have legacy code that has identifiers (variables, methods, etc.) named `assert`…
  - You'll get a compiler error in Java 1.4 onwards
  - The only way to avoid the compiler error is to explicitly tell the java compiler to treat source code as Java 1.3 syntax:

```
javac -source 1.3 MyLegacyCode.java
```

27

This slide is important if you have any legacy code that dates back to Java 1.3 or earlier, where you might have used the word "assert" for some of your variables or methods.



Enabling and Disabling Assertions

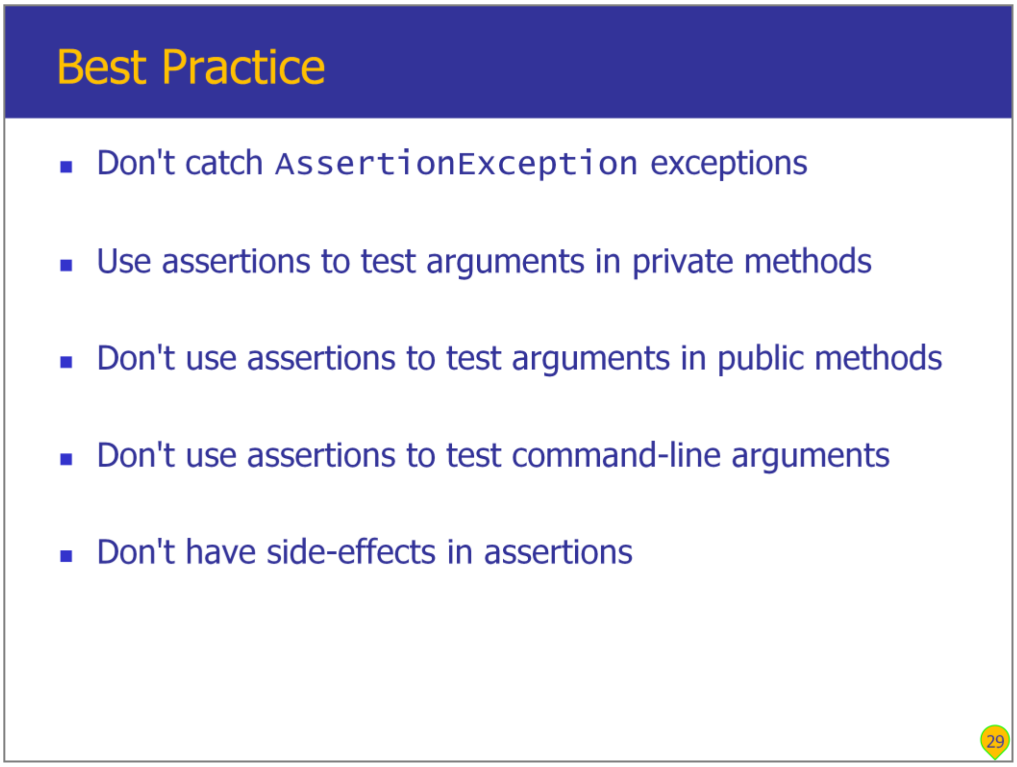
- Assertions are disabled by default
- To enable assertions, specify these JVM arguments:
  - `-ea` Enables assertions for all classes (except system classes)
    `-esa` Enables assertions for system classes
- To disable assertions, specify these JVM arguments:
  - `-da` Disables assertions for all classes (except system classes)
  - `-dsa` Disables assertions for system classes
- Note:
  - `-ea` and `-da` enable you to enable/disable assertions for particular classes and/or packages (and sub-packages)
  - Example:

```
java -ea:mypackage1 -ea:mypackage2 -da:mypackage1.SomeClass MyApp
```

28

If you want assertions to be processed, you must explicitly enable them when you run your Java application. To do this, specify either the `-ea` or `-eas` arguments when you run the JVM.

To disable assertions for particular code packages, specify either the `-da` or `-das` arguments when you run the JVM.



Best Practice

- Don't catch `AssertionException` exceptions
- Use assertions to test arguments in private methods
- Don't use assertions to test arguments in public methods
- Don't use assertions to test command-line arguments
- Don't have side-effects in assertions

29

Here's a bit more info about each of the suggestions on the slide…

Don't catch `AssertionException` exceptions

- Assertions are a development-time aid, not as a run-time feature.

Use assertions to test arguments in private methods

- Helps find bugs in your own code!

Don't use assertions to test arguments in public methods

- Public methods might be called by code you don't control. Who knows what argument values might be passed in! Instead, test for illegal arguments, and throw `IllegalArgumentException`.

Don't use assertions to test command-line arguments

- Same reason as above.

Don't have side-effects in assertions

- Think about this… ☺

Any Questions?