CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Part 1 – Java Generics

Motivation

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Provide a way to re-use the same code with different inputs
- Benefits
  - Stronger type checks at compile time
    - Fixing compile-time errors is easier than fixing runtime errors
  - Elimination of casts
  - Enabling programmers to implement generic algorithms

Prevention of casting

- Without the generics

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- With generics

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

Box class

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

- Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types.
- There is no way to verify, at compile time, how the class is used.
- One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

Generic Version of Box

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

- all occurrences of Object are replaced by T.
- A type variable can be any **non-primitive** type you specify: any class, interface, array, etc.

Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
  - Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

```
Box<Integer> integerBox;
```

- To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

- In Java SE 7 and later, you can invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine the type arguments from the context.
- This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of Box<Integer> with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

Generic Methods

```
public static <T> T best(List<T> entries, ...) { ... }
```

- This says that the best method takes a List of T's and returns a T
- The <T> at the beginning means T is not a real type, but a type that Java will figure out from the method call
- Java will figure out the type of T by looking at parameters to the method call

```
List<Person> people = ...;
Person bestPerson = Utils.best(people, ...);
List<Car> cars = ...;
Car bestCar = Utils.best(cars, ...);
```

Example

```
public class RandomUtils {
   ...

   public static <T> T randomElement(T[] array) {
      return(array[randomIndex(array)]);
   }
}
```

- In rest of method, T refers to a type.
- Java will figure out what type T is by looking at the parameters of the method call.
- Even if there is an existing class actually called T, it is irrelevant here.

This says that the method takes in an array of T's and returns a T. For example, if you pass in an array of Strings, you get out a String; if you pass in an array of Employees, you get out an Employee. No typecasts required in any of the cases.

You can limit T as <T extends Number>
Then, it can be the class (or interface) Number or any of its subclass.

Example

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)
            ++count;
    return count;
}
```

- The greater than operator (>) applies only to primitives.
- You cannot use the > operator to compare objects.
- To fix the problem, use a type parameter bounded by the Comparable<T> interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Correct Code

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

- This restricts the type parameter T to be a type that implements the interface Comparable<T>, guaranteeing that the call to compareTo() is valid.

Generics, Inheritance, and Subtypes

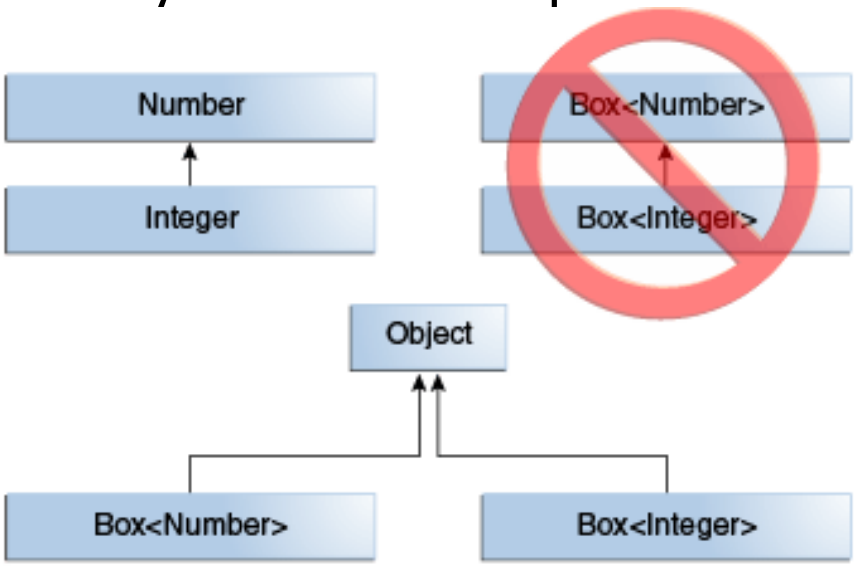
- Integer is also a kind of Number, so the following is valid

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

```
public void boxTest(Box<Number> n) { /* ... */ }
```

- Are you allowed to pass in Box<Integer> or Box<Double>?



Given two concrete types A and B (for example, Number & Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related.

Generic Classes or Interfaces

```
public class ArrayList<E> {

   public E get(int index) { ... }

   public boolean add(E element) { ... }

   ...
}
```

In rest of class, E does not refer to an existing type. Instead, it refers to whatever type was defined when you created the list. E.g., if you did ArrayList<**String**> words = ...; then E refers to String.

This says that get returns an E. So, if you created ArrayList<**Employee**>, get returns an Employee. No typecast required in the code that calls get.

This says that add takes an E as a parameter. So, if you created ArrayList<**Circle**>, add can take only a Circle.

Wildcards

- In generic code, the question mark (?), called the *wildcard*, represents an unknown type
- **Upper Bounded Wildcards**

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

- The upper bounded wildcard, <? extends Foo>, where Foo is any type, matches Foo and any subtype of Foo.
- The process method can access the list elements as type Foo:

```
public static void process(List<? extends Foo> list) {
      for (Foo elem : list) { // ... }
 }
```

Wildcards

- **Lower Bounded Wildcards**
  - A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its *lower bound*: <? super A>.
- You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.
- **Unbounded Wildcards**
  - List<Object> and List<?> are not the same.

What does Effective Java think about Generics?

- Positive things in general!

Type Erasure

- The compiler translates generic and parameterized types by a technique called type erasure.
- After translation by type erasure, all information regarding type parameters and type arguments has disappeared.
- Basically, it omits all information related to type parameters and type arguments.
- For instance, the parameterized type List<String>, List<Long>, … are all translated to type List in the bytecode, which is the so-called *raw type*.

CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Part 2 – Reflection

Reflection

- In computer science, it is the process by which a computer program can observe and modify its own structure and behavior.
- The programming paradigm driven by reflection is called reflective programming.
  - an extension to the object-oriented programming paradigm
  - to add self-optimization to application programs, and
  - to improve their flexibility

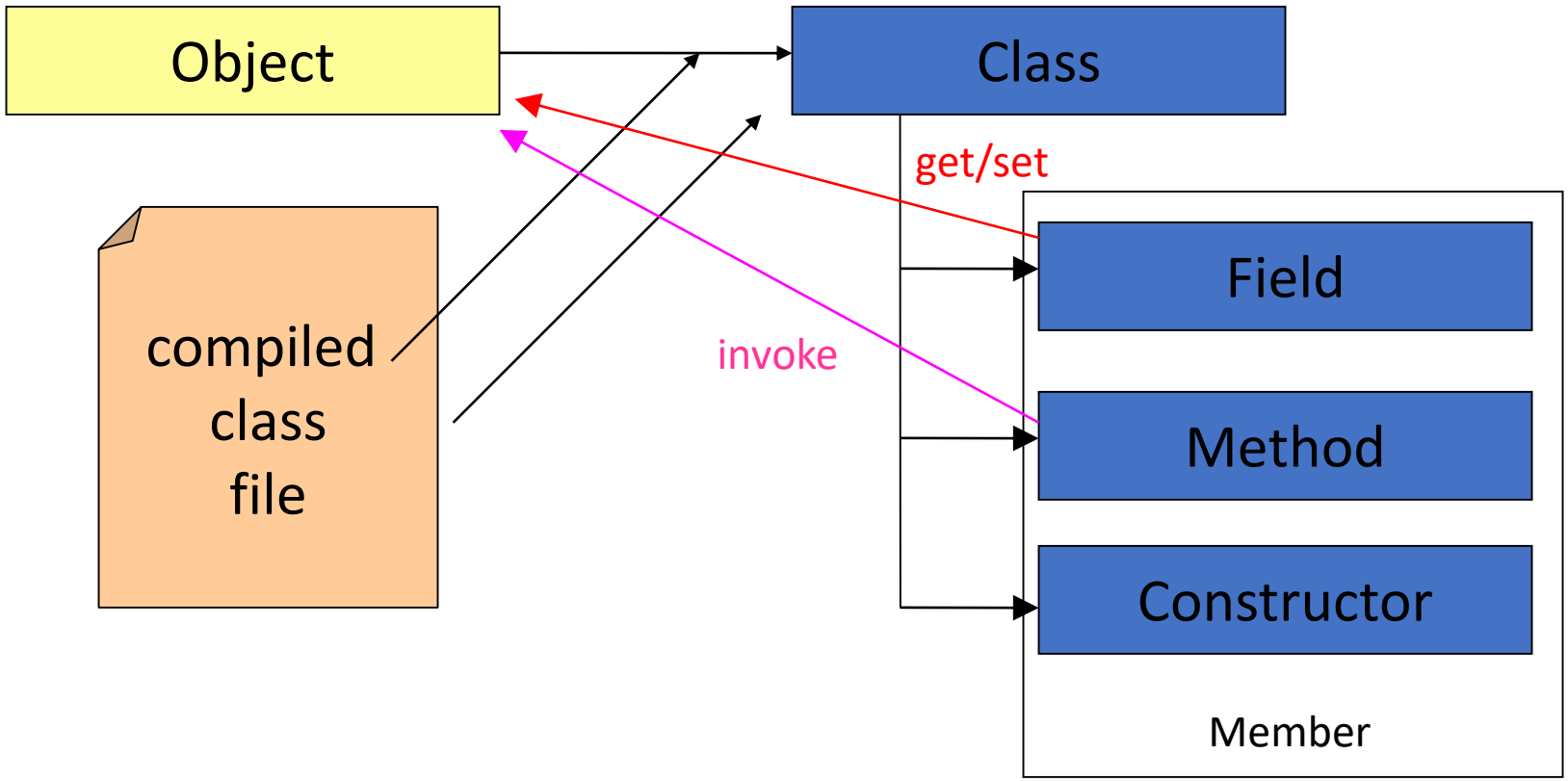
Reflection in Java

```
import java.lang.reflect.*;
```

- Allows you to find out information about any object, including its methods and fields, at run time.
- Used internally by many Java technologies: IDEs, compilers, debuggers, serialization, Java Beans, RMI, ...
- Can also be used to
  - construct new class instances and new arrays
  - access and modify fields of objects and classes
  - invoke methods on objects and classes
  - access and modify elements of arrays

The Class class

- An object of type `Class` represents a Java class.
  - Its fields, methods, constructors, superclass, interfaces, etc.
  - A gateway to the rest of Java's reflection system.



Accessing a Class object

- Ways to get a `Class` object:
  - *If you have an object:* Every object has a `getClass` method to return the `Class` object that corresponds to that object's type.
    - **`Class<Point> pointClass = p.getClass();`**
  - *If you don't have an object, but know the class's name at compile time:* Every class has a static field named `class` storing its `Class` object.
    - **`Class<Point> pointClass = Point.class;`**
  - *If you don't know the type until given its name as a string at runtime:* The static method `Class.forName(String)` will return the `Class` object for a given type; pass it a full class name.
    - **`Class<?> clazz = Class.forName("java.awt.Point");`**

Class class methods

method	description
`getConstructor``(`**`params`**`)` `getConstructors()`	objects representing this class's constructors
`getField``(`**`name`**`)` `getFields()`	objects representing this class's fields
`getInterfaces()`	interfaces implemented by this class
`getMethod``(`**`name, params`**`)` `getMethods()`	objects representing this class's methods
`getModifiers()`	whether the class is public, static, etc.
`getName``()`	full name of this class, as a string
`getPackage()`	object representing this class's package
`newInstance``()`	constructs a new object of this type (if the type has a parameterless constructor)
`toString()`	string matching the class's header

Class class methods 2

method	description
`getAnnotation(`**class**`)` `getAnnotations()`	information about annotations on the class
`getResource(`**name**`)` `getResourceAsStream(`**name**`)`	resource-loading features
`getSuperclass()`	a `Class` object for this type's superclass
`getSimpleName()`	class name without package name
`getTypeParameters()`	all generic type params in this class
`isAnnotation()` `isAnnotationPresent(`**type**`)`	information about annotation types

`isAnonymousClass()`
`isArray(), isEnum()`
`isInterface(),isPrimitive()`
	testing whether the class fits into one of the given categories of types

`isAssignableFrom(`**class**`)`	whether this class is the same as or a supertype of the given class parameter
`getDeclaredFields(), ...`	fields/methods/etc. declared in this file

Reflection example

- Print all the methods and fields in the `Point` class:

```
for (Method method: Point.class.getMethods()) {
    System.out.println("a method: " + method);
}

for (Field field: Point.class.getFields()) {
    System.out.println("a field: " + field);
}
```

Primitives and arrays

- Primitive types and `void` are represented by constants:

constant	alternate form	primitive
`Integer.TYPE`	`int.class`	`int`
`Double.TYPE`	`double.class`	`double`
`Character.TYPE`	`char.class`	`char`
`Boolean.TYPE`	`boolean.class`	`boolean`
`Void.TYPE`	`void.class`	`void`
…	…	…

  - Not to be confused with `Integer.class`, `Double.class`, etc., which represent the wrapper classes `Integer`, `Double`, etc.

- Array classes are manipulated in reflection by static methods in the `Array` class.

Generic Class class

- As of Java 1.5, the Class class is generic: `Class<T>`
  - This is so that known types can be instantiated without casting.

```
Class<Point> clazz = Point.class;
Point p = clazz.newInstance();  // no cast
```

- For unknown types or `Class.forName` calls, you get a `Class<?>` and must still cast when creating instances.

```
Class<?> clazz = Class.forName("java.awt.Point");
Point p = (Point) clazz.newInstance();  // must cast
```

Method class methods

method	description
`getDeclaringClass()`	the class that declares this method
`getExceptionTypes()`	any exceptions the method may throw
`getModifiers()`	whether the method is public, static, etc.
`getName``()`	method's name as a string
`getParameterTypes()`	info about the method's parameters
`getReturnType()`	info about the method's return type
`invoke``(`**`obj, params`**`)`	calls this method on given object *(*`null` *if static)*, passing given parameter values
`toString()`	string matching the method's header

Reflection example 1

- Calling various `String` methods in an Interactions pane:

```
// "abcdefg".length() => 7
> Method lengthMethod =  String.class.getMethod("length");
> lengthMethod.invoke("abcdefg")
7

// "abcdefg".substring(2, 5) => "cde"
> Method substr = String.class.getMethod("substring",
                      Integer.TYPE, Integer.TYPE);
> substr.invoke("abcdefg", 2, 5)
"cde"
```

Reflection example 2

- Calling `translate` on a `Point` object:

```
// get the Point class object; create two new Point()s
Class<Point> clazz = Point.class;
Point p  = clazz.newInstance();
Point p2 = clazz.newInstance();

// get the method Point.translate(int, int)
Method trans =
  clazz.getMethod("translate", Integer.TYPE, Integer.TYPE);

// call p.translate(4, -7);
trans.invoke(p, 4, -7);

// call p.getX()
Method getX = clazz.getMethod("getX");
double x = (Double) getX.invoke(p);        // 4.0
```

Modifier static methods

```
if (Modifier.isPublic(clazz.getModifiers()) { ...
```

static method	description
`isAbstract(`**mod**`)`	is it declared abstract?
`isFinal(`**mod**`)`	is it declared final?
`isInterface(`**mod**`)`	is this type an interface?
`isPrivate(`**mod**`)`	is it private?
`isProtected(`**mod**`)`	is it protected?
`isPublic(`**mod**`)`	is it public?
`isStatic(`**mod**`)`	is it static?
`isSynchronized(`**mod**`)`	does it use the synchronized keyword?
`isTransient(`**mod**`)`	is the field transient?
`isVolatile(`**mod**`)`	is the field volatile?
`toString(`**mod**`)`	string representation of the modifiers such as "public static transient"

Field class methods

method	description
`get``(`**`obj`**`)`	value of this field within the given object

`getBoolean(`**`obj`**`),getByte(`**`obj`**`)`
`getChar(`**`obj`**`),getDouble(`**`obj`**`)`
`getFloat(`**`obj`**`),getInt(`**`obj`**`)`
`getLong(`**`obj`**`),getShort(`**`obj`**`)`
	versions of `get` that return more specific types of data

`getDeclaringClass()`	the class that declares this field
`getModifiers()`	whether the field is private, static, etc.
`getName``()`	field's name as a string
`getType``()`	a `Class` representing this field's type
`set``(`**`obj, value`**`)`	sets the given object's value for this field
`setBoolean(`**`obj, value`**`),` `setByte(`**`obj, value`**`), ...`	versions of `set` that use more specific types of data
`toString()`	string matching the field's declaration

Constructor methods

method	description
`getDeclaringClass()`	the class that declares this constructor
`getExceptionTypes()`	any exceptions the constructor may throw
`getModifiers()`	whether the constructor is public, etc.
`getName()`	constructor's name (same as class name)
`getParameterTypes()`	info about the constructor's parameters
`getReturnType()`	info about the method's return type
`newInstance``(`**`params`**`)`	calls this constructor, passing the given parameter values; returns object created
`toString()`	string matching the constructor's header

Array class methods

static method	description
`get``(`**array, index**`)`	value of element at given index of array

`getBoolean(`**array, index**`),`
`getChar(`**array, index**`),`
`getDouble(`**array, index**`),`
`getInt(`**array, index**`),`
`getLong(`**array, index**`), ...`
	versions of `get` that return more specific types of data

`getLength(`**array**`)`	length of given array object
`newInstance(`**type, length**`)`	construct new array with given attributes
`set``(`**array, index, value**`)`	sets value at given index of given array
`setBoolean(`**array, index, value**`),` `setChar(`**array, index, value**`), ...`	versions of `set` that use more specific types of data

- The `Class` object for array types has a useful method:

static method	description
`getComponentType``()`	a `Class` object for the type of elements

Invocation exceptions

- If something goes wrong during reflection, you get exceptions.
  - Almost all reflection calls must be wrapped in try/catch or throw.
  - Example: ClassNotFoundException, NoSuchMethodError
- When you access a private field, you get an IllegalAccessException.
- When you call a method via reflection and it crashes, you will receive an InvocationTargetException.
  - Inside this is a *nested exception* containing the actual exception thrown by the crashing code.
  - You can examine the nested exception by calling getCause() on the invocation target exception.

AccessibleObject

- Superclass that Field, Method, and Constructor
- **`isAccessible()`**
  - Whether the object can be accessed based on its view type
  - A public field, method, or constructor will return true; the others (e.g. private) will return false.
- **`setAccessible(boolean flag)`**
  - Overrides the accessibility setting to true or false
  - Possible to prevent this using a SecurityManager

The cost of reflection

- **You lose the benefits of compile-time type checking, incl. exception checking**
  - If a program attempts to invoke a nonexistent or inaccessible method reflectively, it will fail at runtime
- **Performance suffers**
  - Reflective method invocation is much slower than normal method invocation
- **The code required to perform reflective access is clumsy and verbose**
  - It is tedious to write and difficult to read

Where to use it

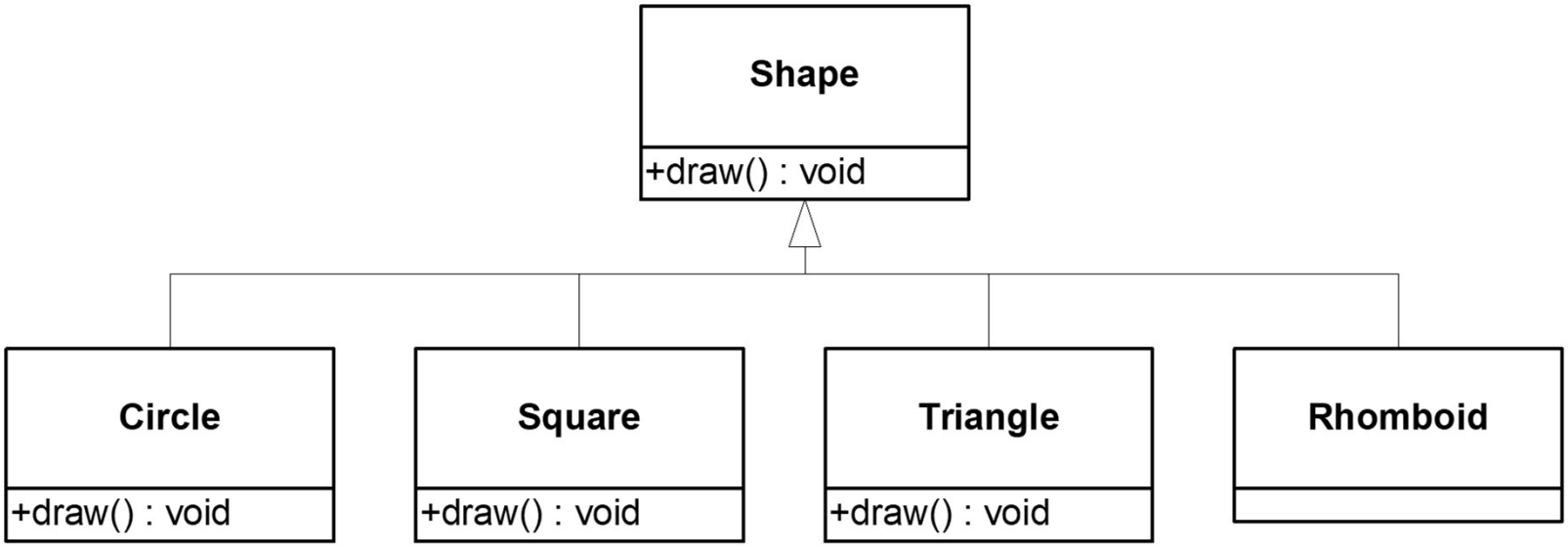
- There are a few sophisticated applications that require reflection.
  - Code analysis tools, dependency injection frameworks, highly dynamic solutions ...

- If you have any doubts as to whether your application requires reflection, it probably doesn't.

Item 65: Prefer interfaces to reflection

Example

- Shape objects to instantiate and manipulate



Factory Method Design Pattern

- **Problem**:
  - How to create shape instances
- **Solution**:
  - Define a method that constructs the object

Factory Method without Reflection

```
public static Shape createShape(String s){
   Shape temp = null;
   if (s.equals("Circle"))
      temp = new Circle();
   else
      if (s.equals("Square"))
         temp = new Square();
      else
         if (s.equals("Triangle")
            temp = new Triangle();
         else
           // …
           // continues for each kind of shape
   return temp;
}
```

- switch-case or cascading if-else statements should scream “redesign me” to the developer
  - Eliminate the switch/case statement
  - Consider a dynamic, better, approach

Factory Method with Reflection

```
public static Shape createShape(String s){
    Shape temp = null;
    try {
        temp = (Shape) Class.forName(s).newInstance();
    }
    catch(Exception e){}
    return temp;
}
```

- Every time you need an instance of some Shape, you have to call `Class.forName(s).newInstance()`

which is really a slow process

- Performance problem when you call it many times!
- Any solution?

Factory Object with Reflection

- Given some factory object (per class) which knows how to create some Shape

```
class SquareFactory extends ShapeFactory
   public Shape createShape(){
      return new Square();
   }
}
```

- Now all you have to do is
  - instantiate only one instance of SquareFactory, and
  - call its createShape method whenever you need some Square
- No such performance problem left!
- But, you need to define a XxxFactory class for each of the Xxx class extending Shape

Factory Object with Reflection

- Instantiating only one instance of XxxFactory

```
public static ShapeFactory createShapeFactory(String s){
   Shape temp = null;
   try {
      temp = (ShapeFactory) Class.forName(s).newInstance();
   }
   catch(Exception e){}
   return temp;
}
```

- Call its createShape method whenever you need some Shape

```
ShapeFactory squareFactory=crateShapeFactory("SquareFactory");
...
Shape square=squareFactory. createShape();
```

CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Part 3 – Exceptions

Exceptions

- The Java programming language uses *exceptions* to handle errors and other exceptional events.
- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Exceptions in Java separates error handling from main business logic
- Java has a uniform approach for handling all errors
  - From very unusual (e.g. out of memory) to more common ones your program should check itself (e.g. index out of bounds)
  - From Java run-time system errors (e.g., divide by zero) to errors that programmers detect and raise deliberately

Throwing and catching

- An error case can throw an exception

```
throw <exception object>;
```

- By default, exceptions result in the thread terminating after printing an error message
- However, exception handlers can catch specified exceptions and recover from error

```
catch (<exception type> e) {
    // statements that handle the exception
}
```

Exceptional flow of control

- Exceptions break the normal flow of control.
- When an exception occurs, the statement that would normally execute next is not executed.
- What happens instead depends on:
  - whether the exception is caught,
  - where it is caught,
  - what statements are executed in the ‘catch block’,
  - and whether you have a ‘finally block’.

Approaches to handling an exception

1. Prevent the exception from happening
2. Catch it in the method in which it occurs, and either
   a. Fix up the problem and resume normal execution
   b. Fix partially and re-throw it *
   c. Handle it then throw a different exception *
3. Do not catch it *
4. With 1 and 2.a the caller never knows there was an error.
5. With 2.b, 2.c, and 3, the caller must handle it.
6. If no one from the method where exception occurred to main catches the exception, the program will terminate and display a stack trace.

* Then, you declare that the method may throw an exception

Catching an exception

```
try {  // statement that could throw an exception
    }
catch (<exception type> e) {
  // statements that handle the exception
}
catch (<exception type> e) { //e higher in hierarchy
  // statements that handle the exception
}
finally {
  // release resources
}
//other statements
```

- At most one catch block executes
- finally block always executes once

Execution of try catch blocks

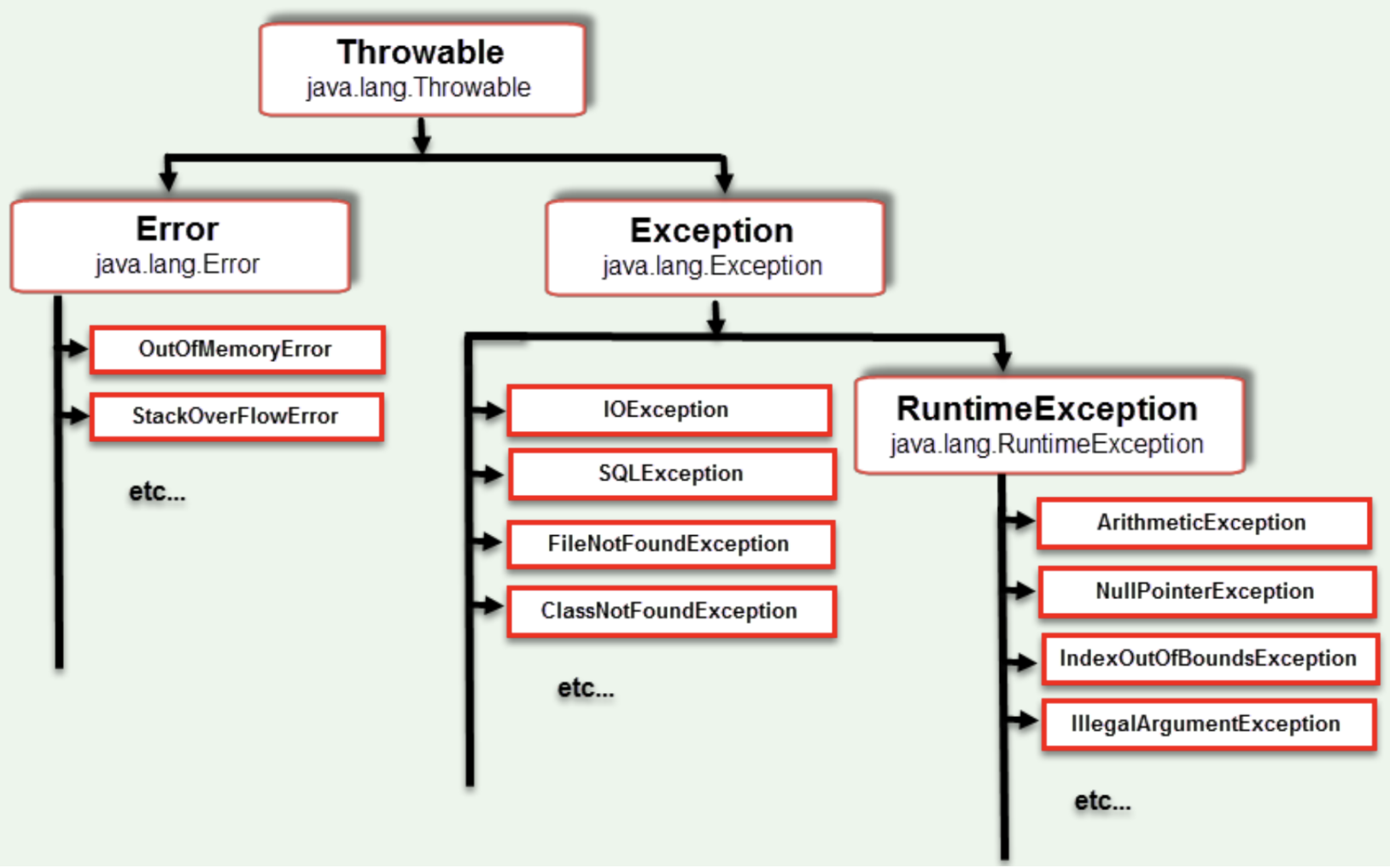
- For normal execution:
  - try block executes, then finally block executes, then other statements execute
- When an error is caught and the catch block throws an exception or returns:
  - try block is interrupted
  - catch block executes (until throw or return statement)
  - finally block executes
- When error is caught and catch block doesn't throw an exception or return:
  - try block is interrupted
  - catch block executes
  - finally block executes
  - other statements execute
- When an error occurs that is not caught:
  - try block is interrupted
  - finally block executes

finally block is always executed unless you call System.exit()

Catch processing

- When an exception occurs, the catch statements are searched top-to-bottom & inner-to-outer order for a catch parameter matching the exception class
- A catch parameter is said to match the exception if it:
  - is the same class as the exception; or
  - is a superclass of the exception; or
  - if the parameter is an interface, the exception class implements the interface.
- The first try/catch statement that has a parameter that matches the exception has its catch statement executed.
- After the catch statement executes, execution resumes with the finally statement, if any, then the statements after the try/catch statement.

Exception hierarchy



Java is strict

- Unlike C++, is quite strict about catching exceptions
- If it is a **checked exception**, Java compiler forces the caller must either catch it or explicitly declare that it may result in an exception.
- By enforcing this, Java guarantees exception correctness at compile time.
- Here's a method that ducks out of catching an exception by explicitly re-throwing it:

```
void f() throws tooBig, tooSmall, divZero { }
```

  - The caller of this method now must either catch these exceptions or declare them in its specification as it may result in them when it calls f().

Checked vs Unchecked

- Checked exceptions are checked at compile-time.
  - if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.
- Unchecked exceptions are not checked at compile time
  - If your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error.
  - Most of the times these exception occurs due to the bad data provided by user during the user-program interaction.
  - It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately during debugging etc.
  - All Unchecked exceptions are direct sub classes of **RuntimeException**.

Example

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
        FileInputStream fis = null;
        /*This constructor FileInputStream(File filename)
         * throws FileNotFoundException which is a checked
         * exception
         */
        fis = new FileInputStream("B:/myfile.txt");
        int k;

        /* Method read() of FileInputStream class also throws
         * a checked exception: IOException
         */
        while(( k = fis.read() ) != -1)
        {
                System.out.print((char)k);
        }

        /*The method close() closes the file input stream
         * It throws IOException*/
        fis.close();
   }
}
```

Will not compile

Example

```
import java.io.*;
class Example {
   public static void main(String args[]) throws IOException
   {
        FileInputStream fis = null;
        /*This constructor FileInputStream(File filename)
         * throws FileNotFoundException which is a checked
         * exception
         */
        fis = new FileInputStream("B:/myfile.txt");
        int k;

        /* Method read() of FileInputStream class also throws
         * a checked exception: IOException
         */
        while(( k = fis.read() ) != -1)
        {
                System.out.print((char)k);
        }

        /*The method close() closes the file input stream
         * It throws IOException*/
        fis.close();
   }
}
```

Option 1 : throw it

Example

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
        FileInputStream fis = null;
        try{
            fis = new FileInputStream("B:/myfile.txt");
        }catch(FileNotFoundException fnfe){
            System.out.println("The specified file is not " +
                        "present at the given path");
         }
        int k;
        try{
            while(( k = fis.read() ) != -1)
            {
                System.out.print((char)k);
            }
            fis.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
         }
   }
}
```

Option 2: handle it

Unchecked Exception Example

```
class Example {
   public static void main(String args[])
   {
        int num1=10;
        int num2=0;
        /*Since I'm dividing an integer with 0
         * it should throw ArithmeticException
         */
        int res=num1/num2;
        System.out.println(res);
   }
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException.

Unchecked Exception Example

```
class Example {
   public static void main(String args[])
   {
        int arr[] ={1,2,3,4,5};
        /* My array has only 5 elements but we are trying to
         * display the value of 8th element. It should throw
         * ArrayIndexOutOfBoundsException
         */
        System.out.println(arr[7]);
   }
}
```

```
class Example {
   public static void main(String args[]) {
        try{
           int arr[] ={1,2,3,4,5};
           System.out.println(arr[7]);
        }
        catch(ArrayIndexOutOfBoundsException e){
           System.out.println("The specified index does not exist " +
                "in array. Please correct the error.");
        }
   }
}
```

It **doesn't mean** that compiler is not checking these exceptions so we shouldn't handle them.

Important notes

- All possible
  - try-catch
  - try-catch-catch…
  - try-catch-finally
  - try-catch-catch-..-finally
  - try-finally
  - Nesting try-catch blocks

```
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}
finally {
    if (input != null) {
       try {
         in.close();
       }catch (IOException exp) {
           System.out.println(exp);
        }
    }
}
```

- The circumstances that prevent execution of the code in a *finally* block are:
  - The death of a Thread
  - Using of the System. exit() method.
  - Due to an exception arising in the finally block.

Example

```
class Example2{
   public static void main(String args[]){
     try{
        System.out.println("First statement of try block");
        int num=45/0;
        System.out.println(num);
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("ArrayIndexOutOfBoundsException");
     }
     finally{
        System.out.println("finally block");
     }
     System.out.println("Out of try-catch-finally block");
   }
}
```

```
First statement of try block
finally block
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

User defined exception in Java

```
class InvalidProductException extends Exception
{
    public InvalidProductException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

public class Example1
{
   void productCheck(int weight) throws InvalidProductException{
        if(weight<100){
                throw new InvalidProductException("Product Invalid");
        }
   }

    public static void main(String args[])
    {
        Example1 obj = new Example1();
        try
        {
            obj.productCheck(60);
        }
        catch (InvalidProductException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
    }
}
```

try-with-resources

```
// try-finally is ugly when used with more than one resource!
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
```

All auto-closed at the end!

```
// try-with-resources on multiple resources - short and sweet
static void copy(String src, String dst) throws IOException {
    try (InputStream   in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

Item 9: Prefer try-with-resources to try-finally