COMP2522

Lesson 2: Inheritance, Polymorphism, Exception Handling

LESSON 2 TOPICS

- Inheritance
- Default constructor
- Overriding
- Substitution and Polymorphism
- Exception handling
- Checked/unchecked exceptions
- Creating your own exceptions

OOP

- Characteristic of object-oriented programming languages:
- Classes and objects
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Composition

- Today we study inheritance and polymorphism

INHERITANCE

- Inheritance allows one class ("subclass" or "child" class) to inherit properties and methods from another class ("superclass" or "parent" class).
- This promotes code reuse and test reuse (more on that later)

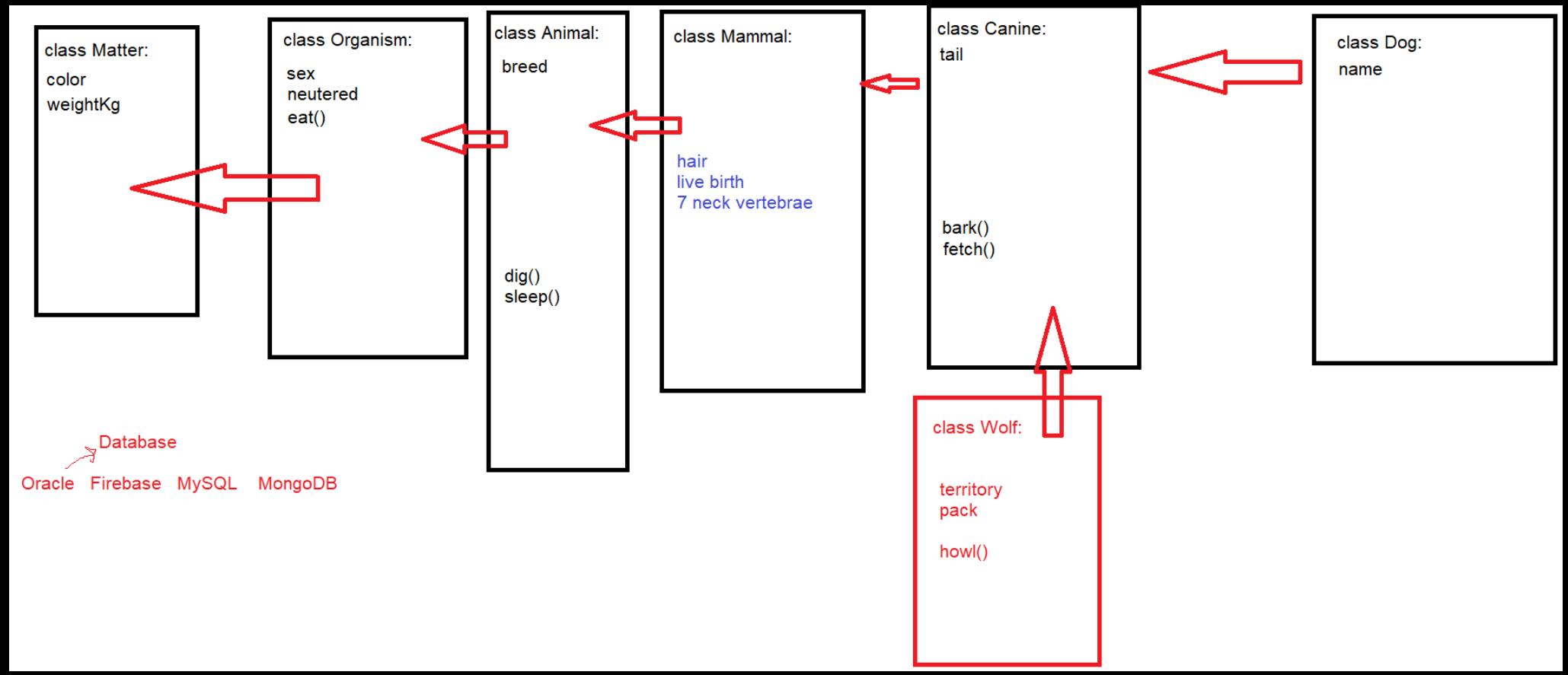
START WITH BIOLOGICAL INHERITANCE

- We are familiar with inheritance in biology
- Imagine behaviors and methods for dogs:
  - color
  - sex
  - breed
  - tail
  - name
  - eat()
  - bark()
  - weight in kg
  - fetch()
  - warm blooded
  - sleep()

MORE CLASSES

- Actually, most of these data and methods are actually common to other classes…they are not unique to dogs
- It is often very useful to consider other classes related to the one we are interested in creating
- If we are creating a Dog class, we would often be likely creating similar classes such as Cat, Wolf, Animal, GermanShepherd, etc…
- If we properly place our data and methods in a hierarchical set of classes, eventually we will avoid lots of duplicated code

EXAMPLE HIERARCHY (PLUS A MORE TECHY ONE)



BENEFITS

- If we took the extra time and extra code to create class Dog, but also Canine, Mammal, Animal, Organism, and Matter…it would be relatively fast and easy to add new classes. Consider:
  - Wolf: another Canine subclass; add howl(), pack, territory, etc…
  - Cat: first add **Feline** subclass of Mammal, then Cat subclass of Feline
  - Salmon: another Animal subclass (Animal → Fish → Salmon?)
  - Pug: a Dog subclass

BENEFITS

- Code and Test Reusability: Allows the reuse of existing code by creating new classes based on existing ones
- Extensibility: New features can be added to existing classes without modifying them (e.g. add child classes, leaving the parent as-is)
- Maintainability: *Centralized* changes to the parent class propagate to all derived classes, simplifying updates and maintenance
- Polymorphism: Enables objects of derived classes to be treated as instances of the parent class, enhancing flexibility and scalability
- Modularity: Promotes modular design by separating common functionality into a base class and specialized behavior into derived classes

MORE EXAMPLES

- Imagine everything in common among the classes Car, Airplane, and Boat
- It is very useful to create a common parent class (superclass): Vehicle

- Imagine a Database class; it would be useful to have that class created first, and then further divided into child classes Oracle, Firebase, MySql, MongoDb, etc….

EXTENDS; OBJECT CLASS

- The *idea* behind inheritance is simple
- The code, in Java, is a bit complicated
- Start by creating the parent class
- In general, the parent class does not know about any child classes it may or may not have
- Use the "extends" keyword for a class to become a child class
- Java supports only single inheritance (can extend maximum one class)
- The root of the class hierarchy is **class Object**: it has no parent
- Every class has Object as a parent (or grandparent, etc…)
- https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Object.html

"EXTENDS" KEYWORD

```
class Animal
{
    private double weightKg;

    Animal(final double weightKg)
    {
        this.weightKg = weightKg;
    }
}
```

```
class Dog extends Animal
{

}
```

Note that the Dog class will not compile, because its constructor is missing something

CONSTRUCTORS IN HIERARCHIES

- If and only if your class does not provide any constructor, Java will (invisibly) create one for you:
- This is called the "default constructor" and it takes zero arguments and does only call super()

- What you typed:

```
class Dog extends Animal
{

}
```

- What Java creates:

```
class Dog extends Animal
{
    Dog()
    {
        super();
    }
}
```

SUPER()

- For most of Java's existence, the very first line in your constructor must be either:
- **super()**
- or
- **this()**
- **…HOWEVER as of Java 22 you can validate your data BEFORE calling super() or this()**
- super() calls the parent constructor; in the previous slide,Dog was passing NO ARGUMENTS to its parent class Animal, but Animal needs "double weightKg" passed to its constructor

- class Dog **extends Animal**

```
{
    Dog(final double weightKg)
    {
        super(weightKg);   // Animal ctor
    }
}
```

- Fixed!

BETTER

- class Animal
  {
  private double weightKg;

  Animal(final double weightKg)
  {
  this.weightKg = weightKg;
  }
  }

- class Dog **extends Animal**
  {
  private final String name;

  Dog(final double weightKg
  final String name)
  {
  super(weightKg);
  this.name = name;
  }
  }

PROTECTED VISIBILITY MODIFIER

- The visibility of private data in a parent class is limited to that class only
- A child can possess parental data that it cannot access directly
- Create getter and setter methods in the parent class so the child can access the data

- The protected visibility modifier limits access to that data/method to:
  - That class itself, and
  - That class's child classes wherever they are, and
  - Any class in the same package

PROTECTED VISIBILITY MODIFIER

- It is a bad idea to make instance variables protected
- Trust your own code in your own class
- Do not allow any class – child or otherwise – to have direct access
- Make setters and getters protected instead

- Bad:
- class Animal
  {
  protected double weightKg;
  Animal(final double weightKg)
  {
  this.weightKg = weightKg;
  }
  }

- Good:
- class Animal
  {
  private double weightKg;
  Animal(final double weightKg)
  {
  this.weightKg = weightKg;
  }
  // provide **protected** getters and setters
  }
  }

DEEPER HIERARCHY


```
class Animal
{
    private double weightKg;
    Animal(final double weightKg)
    {
        this.weightKg = weightKg;
    }
    protected double getWeightKg()
    {
        return weightKg;
    }
    protected void setWeightKg(final double kg)
    {
        this.weightKg = kg;
    }
}
```

```
class Dog extends Animal
{
    private final String name;

    Dog(final double weightKg,
            final String name)
    {
        super(weightKg);
        this.name = name;
    }

    protected String getName()
    {
        return name;
    }
}
```

```
public class Pitbull extends Dog
{
    private boolean trained;

    Pitbull(final double weightKg,
            final String name,
            final boolean trained)
    {
        super(weightKg, name);
        this.trained = trained;
    }

    protected boolean isTrained()
    {
        return trained;
    }

    protected void setTrained(final boolean trained)
    {
        this.trained = trained;
    }
}
```

POLYMORPHISM AND SUBSTITUTION

- Polymorphism refers to the ability to treat objects of different child ("derived") classes through a common interface, typically the parent class, allowing methods to behave differently based on the object's actual type at runtime
- This enables flexibility, where different objects can respond to the same method call in their own way
- Substitution (aka "the Liskov Substitution Principle") specifically refers to the idea that objects of a subclass should be able to replace objects of the parent class without affecting the correctness of the program

- **I.e.: A child-class object can be used wherever a parent class object is expected**

SUBSTITUTION

```
class Dolphin extends Animal
{
    private boolean inCaptivity;

    Dolphin(final double weightKg,
                final boolean inCaptivity)
    {
        super(weightKg);
        this.inCaptivity = inCaptivity;
    }

    protected boolean isInCaptivity()
    {
        return inCaptivity;
    }
}
```

```
class Zoo
{
    public static void main(final String[] args)
    {
        final Animal a1;
        final Animal a2;
        final Animal a3;
        final Animal a4;

        a1 = new Animal(200.0);
        a2 = new Dog(30.0, "Rex");
        a3 = new Pitbull(50.0, "Rocky", true);
        a4 = new Dolphin(300.0, false);
    }
}
```

- The "static type" of all four animals is set at compile time.
- In this case, all of a1, a2, a3, and a4 have a static type of "Animal".

- The "dynamic type" is set at runtime.
- In this case, the dynamic type of a1 is Animal; a2 is Dog; a3 is Pitbull; a4 is Dolphin.

- **A child can always be used in place of a parent in Java.**

INSTANCEOF

- The "instanceof" operator is not used often.
- It will return true if an object is of the specified class, or is some child (grandchild, etc…) of it.

```
final Animal a3;

a3 = new Pitbull(50.0, "Rocky", true);

System.out.println(a3 instanceof Object);  // true for objects of any class
System.out.println(a3 instanceof Animal);  // true
System.out.println(a3 instanceof Dog);     // true
System.out.println(a3 instanceof Pitbull);  // true
System.out.println(a3 instanceof Dolphin); // false
```

CASTING

- In Java, we can convert between compatible types using CASTING
- Put the new type in parentheses before the data itself
- We can use child-class-specific methods this way
- E.g. System.out.println((double)4); // change 4 to a double; prints 4.0

```
final Animal a3;
a3 = new Pitbull(50.0, "Rocky", true);
 if(a3 instanceof Dog)
{
   final Dog d;
   d = (Dog)a3;   // cast
   d.bark();         // treat as Dog
}

if(a3 instanceof Dolphin)
{
   final Dolphin d;
   d = (Dolphin)a3;   // cast
   d.swim();             // treat as Dolphin
}
```

- You often should check the object's class before casting
- Casting to an incompatible class will result in a ClassCastException:

- final Dolphin d1;
- final Animal a4;

- a4 = new Dog();
- d1 = (Dolphin)a4;   // compiles, but…

- d1.swim();   // crashes

GETCLASS()

```java
package ca.bcit.comp2522.zoo;
class Zoo
{
    public static void main(final String[] args)
    {
        final Animal a1;
        final Animal a2;
        final Animal a3;
        final Animal a4;

        a1 = new Animal(200.0);
        a2 = new Dog(30.0, "Rex");
        a3 = new Pitbull(50.0, "Rocky", true);
        a4 = new Dolphin(300.0, false);

        System.out.println(a1.getClass());                // class ca.bcit.comp2522.zoo.Animal
        System.out.println(a2.getClass().getName());      // ca.bcit.comp2522.zoo.Dog
        System.out.println(a3.getClass().getSimpleName()); // Pitbull
```

- The Object class has methods available to absolutely all classes, since all classes (including arrays) automatically extend Object
- The Object class has methods that may be used in place of the instanceof operator: getClass() and getName() and getSimpleName() (no package)

OVERRIDING / POLYMORPHISM

- Method overriding is not the same as overloading
- Overriding means a child (or grandchild, etc…) class redefines a method that was already defined higher in the hierarchy
- The signature (method name, parameters) must be the same in the child as in the parent
- Use the @Override annotation in the new version of the method; if you are in fact NOT overriding a method, the compiler will tell you
- Static methods and private methods cannot be overridden
- **The new method's visibility must be the same or wider than the parent version**

@OVERRIDE

- Add these methods to the Animal class:

```
public void speak()
{
    System.out.println("Speaking...");
}
protected void move()
{
    System.out.println("Moving...");
}
```

- Now every Animal can speak and move
- Since Animals do these behaviors differently, override them in the child classes
- speak() MUST remain public in the child versions
- move() must be **protected or public**, or it won't compile
- This is better and more flexible than having specific methods such as bark() and swim() in *only some of the classes*

Dog:

```
@Override
    public void speak()
    {
        System.out.println("woof woof");
    }
    @Override
    protected void move()
    {
        System.out.println("running...");
    }
```

Dolphin:

```
 @Override
    public void speak()
    {
        System.out.println("Eee ee e");
    }
    @Override
    public void move()
    {
        System.out.println("swimming...");
    }
```

```
a1 = new Animal(200.0);
a2 = new Dog(30.0, "Rex");
a3 = new Pitbull(50.0, "Rocky", true);
a4 = new Dolphin(300.0, false);
```

Polymorphism:
At runtime the "closest-matching" version of the overridden methods are called

```
a1.speak(); // Speaking...
a1.move();  // Moving...

a2.speak(); // woof woof
a2.move();  // running...

a3.speak(); // woof woof
a3.move();  // running...

a4.speak(); // Eee ee e
a4.move();  // swimming...
```

Note that we can override methods again and again. If Pitbull *further overrode* these methods, then a3 (above) would have called the Pitbull's version instead

JAVA EXCEPTIONS

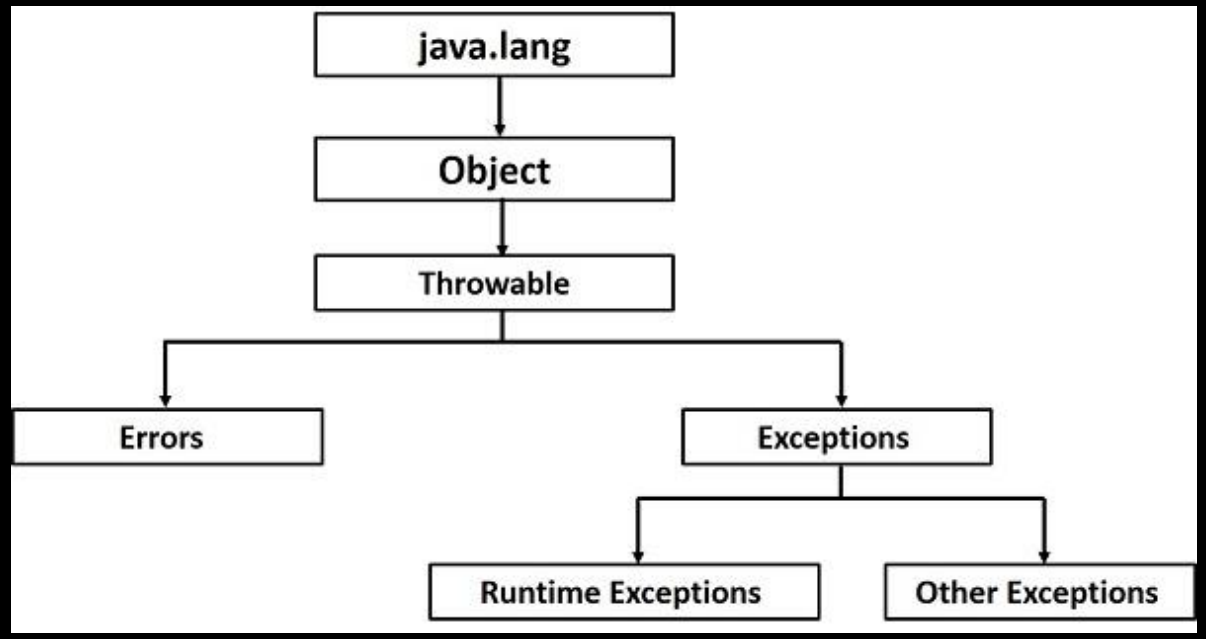
- There are various "categories" of Exceptions that can be thrown
  - **Checked** Exceptions: 1) *must* be declared; 2) *must* be wrapped in try/catch
  - **Unchecked** Exceptions: *optionally* can be in try/catch (aka RuntimeException)
  - Custom Exceptions: make your *own* (checked or unchecked) Exceptions
- Use checked exceptions when things are likely to go wrong, and can be handled: this *forces* other code to use try/catch and forces the thrower to declare that it throws; both of these are checked at compile time
- Use unchecked exceptions when things are unlikely to go wrong
- Use custom exceptions only if they add lots of readability and value; otherwise stick to the normal Java Exception classes

CUSTOM EXCEPTIONS

- A custom Exception class can help add readability
- Good idea: IllegalChessMoveException
- Bad idea: IllegalLastNameException

- Use IllegalArgumentException instead of IllegalLastNameException
- Java developers are used to the IllegalArgumentException
- IllegalLastNameException doesn't add much more information
- IllegalChessMoveException (in a Chess class) could be helpful though

JAVA EXCEPTIONS



- R**un**timeExceptions are **un**checked (no need for *try/catch*; no need to declare with *throws* declaration)

CUSTOM EXCEPTIONS

- Classes that extend RuntimeException are unchecked:
  - no try/catch needed
  - no throws declaration needed
- The following custom Exception class extends Exception (not RuntimeException), so it is a *checked* Exception:
  - try/catch is mandatory (or the class will not even compile)
  - throws declaration is mandatory (or the class will not even compile)
  - Also let's use the @throws Javadoc tag too, from the method/constructor that throws it

```
class IllegalChessMoveException
                extends Exception    // not "extends RuntimeException" (which would be unchecked)
{
   IllegalChessMoveException(final String message)
   {
      super(message);
   }
}
```

CHECKED EXCEPTION: MUST DECLARE IT THROWS

```
class Pawn
{
    public void move(char startFile, int startRow, char endFile, int endRow)
    {
        if(endRow <= startRow)
        {
            throw new IllegalChessMoveException("illegal pawn move");
        }
    }
}
```

Unhandled exception: IllegalChessMoveException

Add exception to method signature Alt+Shift+Enter More actions... Alt+Enter

IllegalChessMoveException
IllegalChessMoveException(String message)

CHECKED EXCEPTION: MUST DECLARE IT THROWS

```
class Pawn
{
    /**
     * @param startFile the file from which the pawn moves
     * @param startRow the row from which the pawn moves
     * @param endFile the file to which the pawn moves
     * @param endRow the row to which the pawn moves
     * @throws IllegalChessMoveException if the pawn move is illegal
     */
    public void move(char startFile, int startRow, char endFile, int endRow) throws IllegalChessMoveException
    {
        if(endRow <= startRow)
        {
            throw new IllegalChessMoveException("illegal pawn move");
        }
    }
}
```

CHECKED EXCEPTION: MUST TRY/CATCH

- Note that this will not compile unless we:
- 1) wrap it in a try/catch, or
- 2) change IllegalChessMoveException to unchecked by changing its parent to RuntimeException

```
public class Chess
{
    public static void main(String[] args)
    {
        Pawn pawn = new Pawn();
        pawn.move( startFile: 'a',  startRow: 2,  endFile: 'a',  endRow: '7');
    }
}
```

Unhandled exception: IllegalChessMoveException

Add exception to method signature Alt+Shift+Enter More actions... Alt+Enter

```
Pawn
public void move(char startFile,
                 int startRow,
                 char endFile,
                 int endRow)
throws IllegalChessMoveException
```

Throws: IllegalChessMoveException

CHECKED EXCEPTION: MUST TRY/CATCH


```
public class Chess
{
    public static void main(String[] args)
    {
        Pawn pawn = new Pawn();
        try
        {
            pawn.move( startFile: 'a',  startRow: 2,  endFile: 'a',  endRow: '7');
        }
        catch(IllegalChessMoveException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

- It compiles now

UNCHECKED: TRY/CATCH OPTIONAL

```
class IllegalChessMoveException extends RuntimeException
{
    IllegalChessMoveException(String message)
    {
        super(message);
    }
}
```

```
class Pawn
{
    public void move(char startFile, int startRow, char endFile, int endRow)
    {
        if(endRow <= startRow)
        {
            throw new IllegalChessMoveException("illegal pawn move");
        }
    }
}
```

```
public class Chess
{
    public static void main(String[] args)
    {
        Pawn pawn = new Pawn();
        pawn.move( startFile: 'a',  startRow: 2,  endFile: 'a',  endRow: '7');
    }
}
```

move() **throws** declaration is also optional
Usually, it's a bad idea to treat an unchecked Exception like a checked one

CLASS HIERARCHY

- Throwable: anything with this parent, can be thrown
- Error: the JVM that runs Java programs has a problem with hardware or software (e.g. out of memory)
- Exception: program-specific problem out of the programmer's CHECKED control (e.g. a necessary file has been deleted; a network connection to a remote database is broken)
- RuntimeException: UNCHECKED a problem made by the programmer needs to be fixed (e.g. tried to call toUpperCase() on a null String reference): any resulting crash is good because it draws our attention and we will fix the coding error.

EXCEPTION HANDLING

- main() should not throw exceptions
- The finally{ } block is always executed (even if try/catch blocks *return*)
- The compiler forces us to handle checked exceptions (by try/catch or by declaring and rethrowing them)
- The compiler does not force us to handle unchecked exceptions; don't try/catch these; let them crash the program and thus draw our attention to make repairs
- To handle an exception from a catch block, we can do a number of different things:
  - Handle it directly in that catch block
  - Call a handler method from the catch block
  - Print the stack trace in the catch block (ok for development, not for production)

CUSTOM EXCEPTIONS

- We can extend the Exception class to create custom Exception classes or even hierarchies (i.e. multiple Exception classes with our own custom parent class)
- The msg string should be meaningful (e.g. “number 130 out of range of 1 – 100”)