Week 2

Friday

Agenda

- Review the review assignment
- More about Exceptions and throwing
- Inheritance, part one
- Talk about the next assignment (SheridanMaps)

Advanced Exception Handling

Creating Exception classes, try/catch, throws, throw, and finally

Chapter 14.9: Exception handling

- Exceptions are expected and unexpected
  - Exceptions describe ‘problems’ that occur
  - Expected can be built in your code
  - Unexpected handle the ‘what-if’ scenarios
- Example:
  - Expected: logging in to Slate2 and typing the wrong password

```
public void connectToSlate2(String user, String password){
  while(!isValidPassword(user, password)) {
    System.out.println(“Sorry, wrong password”); ...
```

  - Unexpected: logging in to Slate2 and the server is down

```
public void connectToSlate2(String user, String password){
  ====> java.lang.ServerConnectionException
```

Chapter 14.9: Exception handling

- try/catch: turn the unexpected into the expected
  - "try" prepares your code for any possible unexpected problems
  - "catch" captures those problems and handles them
  - Use one catch block for each exception, or a generic catch to capture all exceptions

```
try
{
     connectToSlate2(user, pass);
}
catch(ServerConnectionException scx)
{
     handleServerDown();
}
catch(AccountLockedException alx){
     handleLockedAccount();
}
catch(AccountRemovedException arx) {
     handleAccountRemoved();
}
```

Useful for specific handling

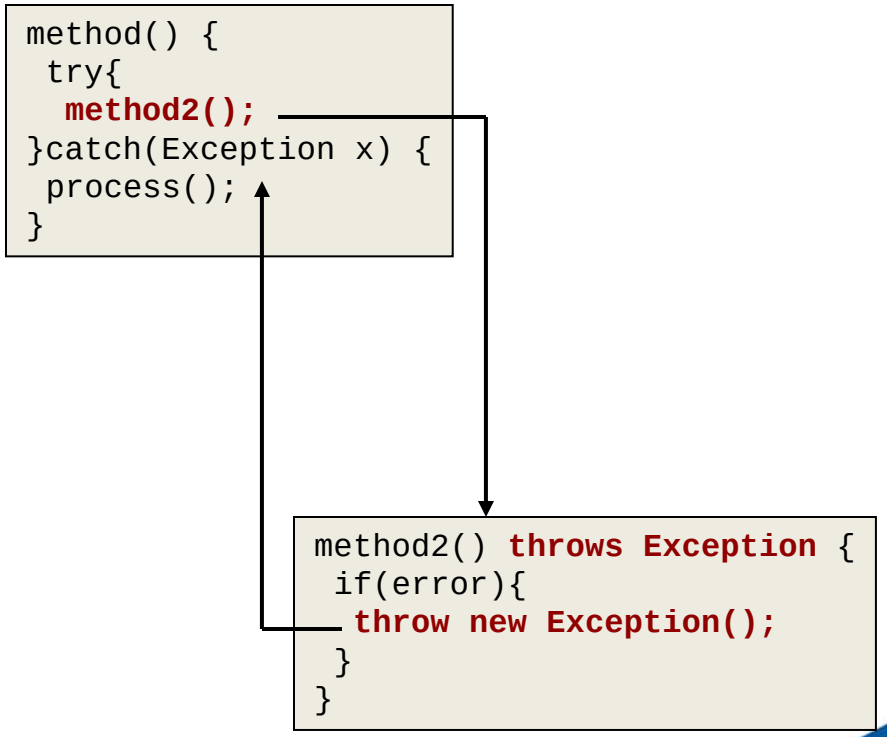
```
try
{
     connectToSlate2(user, pass);
}
catch(GenericException scx)
{
     handleAnyException();
}
```

Useful for catching everything

Sheridan

Chapter 14.9: Exception handling

- Where do exceptions come from?
  - From other code, of course!
- Declare exceptions
  - *throws*: it's a warning in the method signature - my method may result in an exception
  - *throw*: that error has occurred, sound the alarm
- Other things to know:
  - Re-throw: You can throw an exception in a catch block
  - Call stack: a breadcrumb trail, where exactly was I when this crashed?
  - *finally*: a reserved word in Java, executes a block of code even if there is an exception



Chapter 14.9: Exception handling

- Build your own Exception class

```
public class JustinBieberException extends Exception {
   public String getMessage() {
      return "Really?  What were you thinking.";
   }
}
```

"is a type of" Exception

Exception classes are built like any other class, but they're a type of Exception

```
public class Song {
   private String title;
   private String artist;

   public void addSong() throws JustinBieberException {
      if(getArtist().equals("Justin Bieber")) {
         throw new JustinBieberException();
      }
   }
}
```

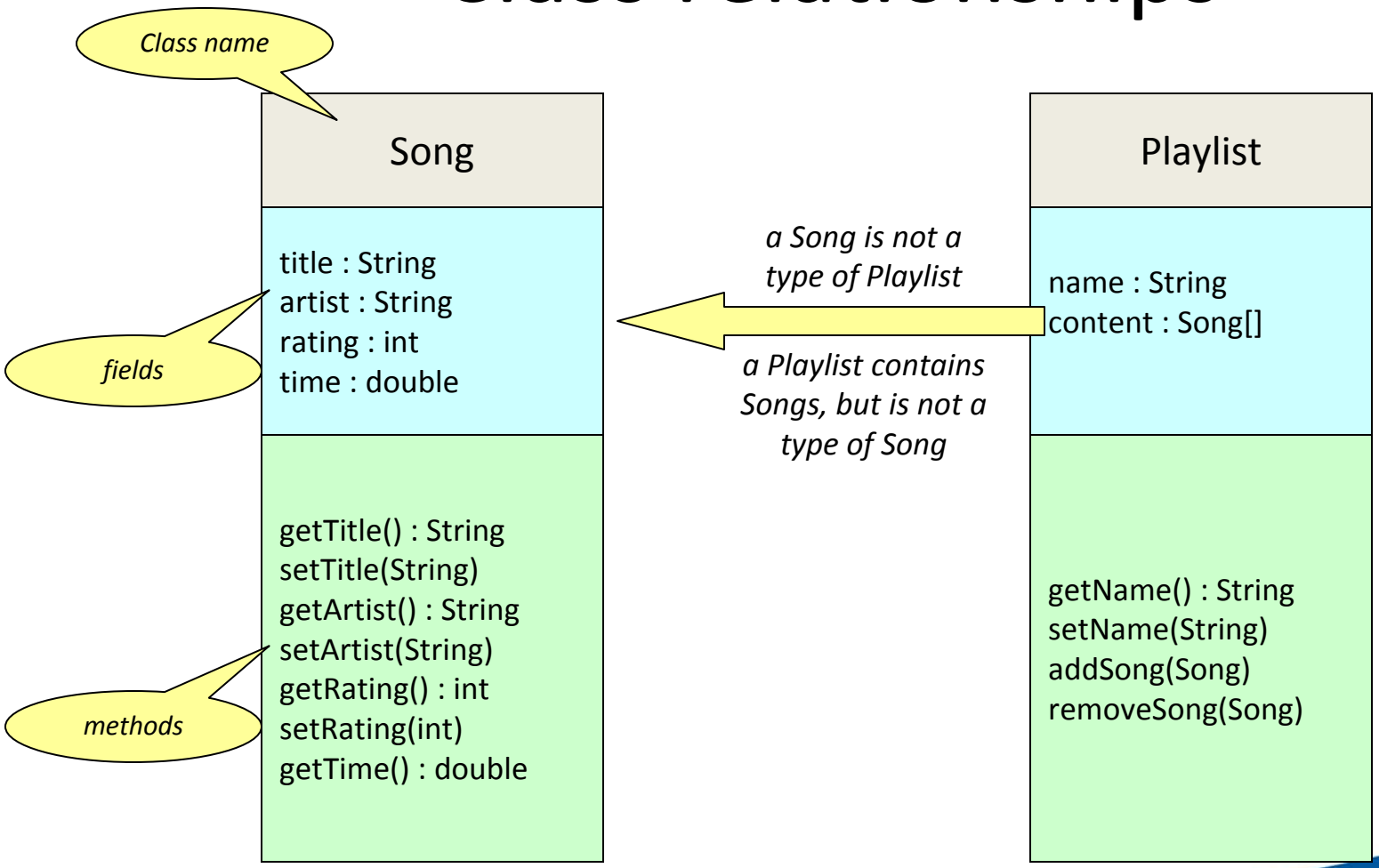
Use the exception in any other method

Constructor for a new Exception

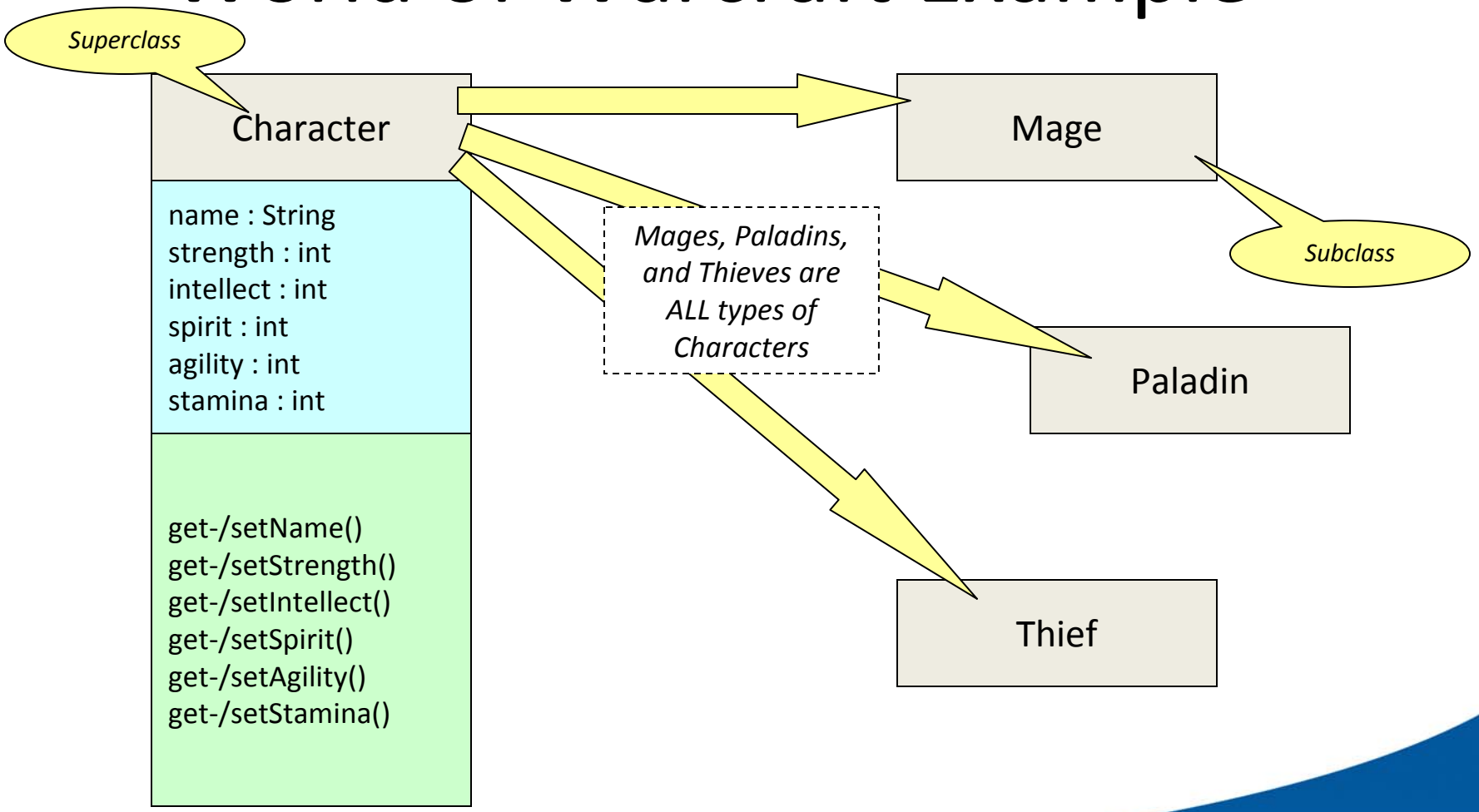
Inheritance

Superclass, subclass, this/super

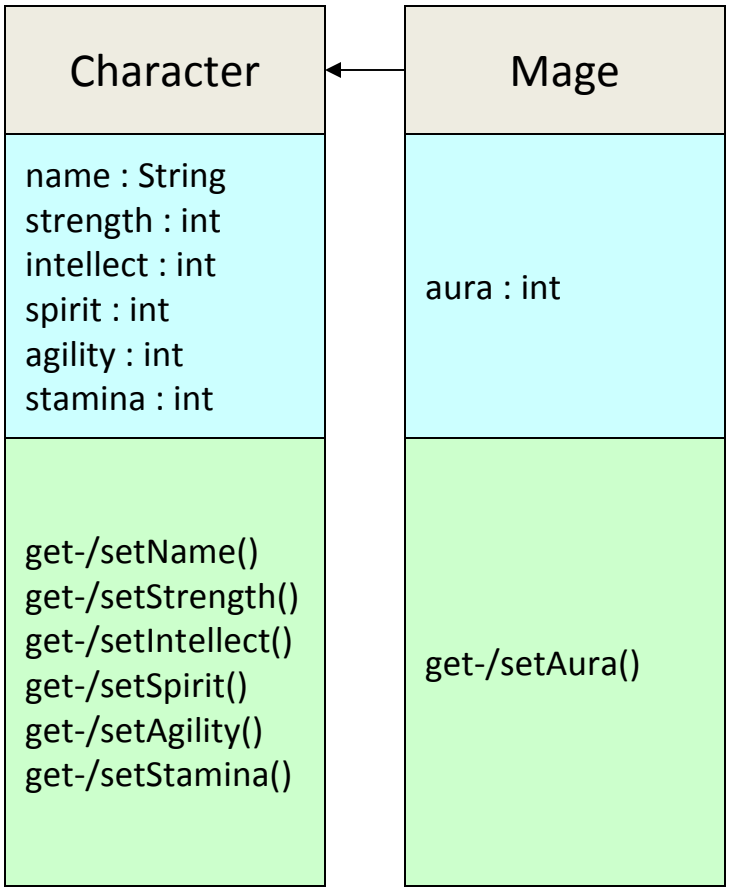
Class relationships



World of Warcraft Example



World of Warcraft Example



fields

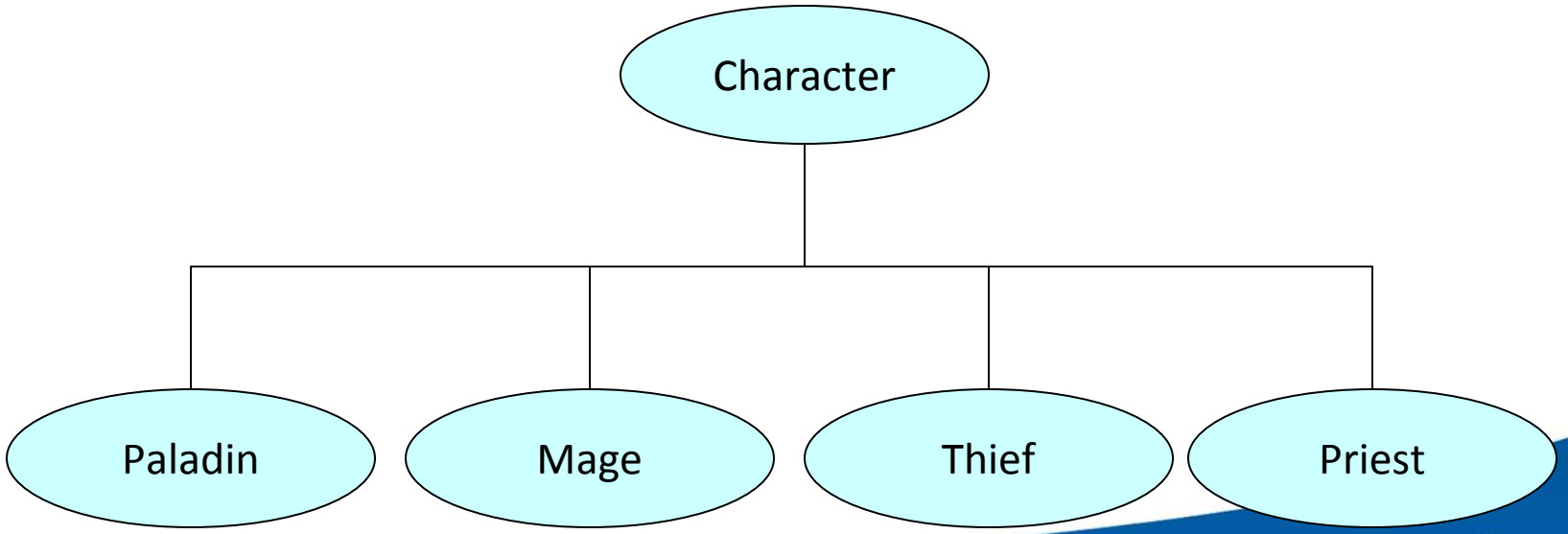
```
public class Mage extends Character {
   private int aura;

   public Mage() {
      super();
   }
   public Mage(String name) {
      super(name);
   }

   public int getAura() { ... }
   public void setAura(int a) { ... }

}
```

Chapter 11.1: Inheritance

- Super and subclasses
  - The *super* keyword is used to specifically call the parent class
    - It can be used in a method or constructor
    - It's not always necessary, however
  - Subclasses tend to be more specific than superclasses
    - "Mage is a specific type of Character"
    - "Circle is a specific type of GeometricShape"
  - Should always represent an "is-a-kind-of" relationship
  - Some languages allow multiple inheritance, but not Java

Chapter 11.1-11.3: Inheritance

- Why extend classes?
  - Express classes in parent-child (superclass/subclass) relationships
  - Reuse methods and fields between classes
  - Reusability is very important in O/O!



Chapter 11.1-11.3: Inheritance

- How inheritance works:
  - Mage is type of Character (Mage is subclass of Character)
  - All Characters have a name, intellect, and other properties

```
public void exploreDungeon() {
   Mage gandalf = new Mage("Gandalf");

   gandalf.setIntellect(22);
   gandalf.setAura(15);

   ...
```

Mage does not have setIntellect(int) method – it's inherited from superclass

Mage has setAura(int) method – unique to the Mage class (Thief cannot call setAura())

Chapter 11.4: Overriding, Overloading

- Methods in super and subclass
  - If a method cannot be found in the subclass, Java looks for it in the parent
    - paladin.getName() does not exist, but Paladin is a type of Character and Character has getName()
  - Methods can only be discovered in the parent class if they're *visible*
    - *private* methods cannot be seen, *public* methods can
    - We'll talk about *protected* soon...
- Overriding and Overloading
  - If a method is EXACTLY the same in the subclass, it *overrides* the parent
    - e.g.: Mage can define a setIntellect(int) method which overrides the superclass
  - Static methods are never overridden
  - If a method has the same name but different signature, it *overloads* the parent
    - e.g.: Mage can define setIntellect(double) – the parameter in the signature is different, so now Mage has two overloaded setIntellect() methods available

How to design good classes

- Cohesion:
  - Describe a single entity and all operations fit in that entity
    - e.g.: Song class has operations such as editSong() but not editPlaylist()
- Consistency
  - Use standard naming convention
  - Create classes with at least two constructors (argument/no argument)
- Encapsulation
  - Fields should be private, use getters and setters to work with values
- Clarity
  - Independent methods
  - Variables and methods are easy to explain
  - The class, methods, and fields “make sense”

```
public class Song {
    private String title;
    private String artist;
    private int rating;
    private double time;

    public Song() {
    }

    public Song(String name) {
        this.setTitle(name);
    }

    public String getTitle() {
        return title;
    }
    public void setTitle(String name) {
        this.title = name;
    }
}
```

More hints about classes

- Classes provide abstraction
  - Methods describe how the class is used
    - playSong(), editSong(), addSongToPlaylist(Playlist)
  - Provides collection of methods and fields, not just one or two
  - The class' fields and methods form the *class contract*
- Other stuff
  - Objects can be immutable (unchangeable)
    - Set with constructors, but have no setter methods
  - *this* and *super*
    - Keywords which allow you to be specific when working with classes (either *this* class specifically, or its parent, *super*)
    - Helps to avoid ambiguity when referring to classes

Things to think about...

- In World of Warcraft example... how would you model this?
  1. Characters can be part of a Faction: Alliance or Horde. Humans, Dwarves, and Gnomes are all part of the Alliance. Goblins, Trolls, and the Undead are all part of the Horde.
     - Think of get-/setFaction() methods
  2. Every character has a maximum Health and Energy amount, which are functions of a character's attributes. High Strength and Stamina give greater Health, Spirit and Intellect give greater Energy.
     - Think of get-/setHealth() and get-/setEnergy() methods
     - Think of get-/setCurrentHealth() and get-/setCurrentEnergy() methods
  3. Every Character also has a Race. There are Humans, Dwarves, Gnomes, Orcs, Undead, Tauren, Trolls, and more.
     - You can be a Gnome Thief or a Troll Priest.
     - Where does Race fit in with Character and its subclasses?