# COMP 110/L Lecture 25

Kyle Dewey

# Outline

- Overloading
- Exceptions

# Overloading

# Recall

```
Random r = new Random();
r.nextInt();
```

-You've seen this sort of use of Random...

# Recall

```
Random r = new Random();
r.nextInt();
```

```
Random r = new Random(1231);
r.nextInt(42);
```

- ...you've also seen this variant of Random
- These coexist

# Overloading

Two methods/constructors can have the same name in the same scope, as long as their *signatures* differ

-A signature consists of both the name and the input types
-As such, as long as two methods take different inputs, they may have the same name (while the names in the signatures are the same, the inputs differ, so the signatures are overall different)

# Overloading

Two methods/constructors can have the same name in the same scope, as long as their *signatures* differ

```
public class Random {
   public Random() { ... }
   public Random(long seed) { ... }

   public int nextInt() { ... }
   public int nextInt(int i) { ... }
}
```

-A signature consists of both the name and the input types
-As such, as long as two methods take different inputs, they may have the same name (while the names in the signatures are the same, the inputs differ, so the signatures are overall different)

# Example:

# `BasicOverloading.java`

# Example

- `OverloadingBase.java`
- `OverloadingSub.java`
- `OverloadingBaseSub.java`

# Overloading with Polymorphism

Method is chosen based on *compile-time* type

# Overloading with Polymorphism

Method is chosen based on *compile-time* type

# Example

- `OverloadingBase.java`
- `OverloadingSub.java`
- `OverloadingAdvanced.java`

# Overloading vs. Overriding

- Overloading based on compile-time types
- Overriding based on run-time types
  - **Runtime type of** `base` **is** `Sub`:

```
Base base = new Sub();
```

# Exceptions

# Recall

```
int[] array = new int[3];
int result = array[27];
```

-What happens if this code snippet is run?

# Recall

```
int[] array = new int[3];
int result = array[27];
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
```

-What happens if this code snippet is run?

# Recall

```
int[] array = new int[3];
int result = array[27];
```

---

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
```

---

```
int result = Integer.parseInt(“hello”);
```

-What happens if this code snippet is run?

# Recall

```
int[] array = new int[3];
int result = array[27];
```

---

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
```

---

```
int result = Integer.parseInt(“hello”);
```

---

```
Exception in thread "main"
java.lang.NumberFormatException
```

-What happens if this code snippet is run?

# Exceptions

- Intended to signal events which happen infrequently but cannot be ignored
    - “Exceptional”
    - Errors are common examples
- Can define different kinds of exceptions for different conditions

# Exceptions

- Intended to signal events which happen infrequently but cannot be ignored
    - “Exceptional”
    - Errors are common examples
    - Can define different kinds of exceptions for different conditions

```
java.lang.ArrayIndexOutOfBoundsException
    java.lang.NumberFormatException
```

-For example, we can define exceptions for an array index being out of bounds (one kind of error condition), and exceptions indicating that a number was of an unexpected format / we couldn’t parse it (another kind of error condition)

# Defining Exceptions

Inherit from the `Exception` class.

Both a no-arg constructor and one that takes a `String`.

-The passed String indicates a message which can encode more details (e.g., “57 is not negative”)

# Defining Exceptions

Inherit from the `Exception` class.

Both a no-arg constructor and one that takes a `String`.

```
public class MyException
   extends Exception {
   public MyException(String message) {
      super(message);
   }
}
```

-The passed String indicates a message which can encode more details (e.g., “57 is not negative”)

# Example:

# `MyException.java`

# Throwing Exceptions

Methods must state which exceptions they throw, using the `throws` reserved word

# Throwing Exceptions

**Methods must state which exceptions they throw, using the `throws` reserved word**

```
public static void myMethod()
   throws MyException {
   ...
}
```

-Declaring that myMethod throws MyException

# Throwing Exceptions

**Methods must state which exceptions they throw, using the `throws` reserved word**

```
public static void myMethod()
   throws MyException {
   ...
}
```

```
public static void myMethod()
   throws MyException, OtherException {
   ...
}
```

-Declaring that myMethod throws MyException or OtherException

# Throwing Exceptions

Exceptions can be thrown with the `throw` reserved word

# Throwing Exceptions

Exceptions can be thrown with the `throw` reserved word

```
public static void myMethod()
   throws MyException {
   if (...) {
      throw new MyException(“message”);
   }
}
```

# Example

- `MyException.java`
- `ThrowMyException.java`

-Key point in the example: thrown exceptions can traverse method boundaries. Main can also throw MyException even though it doesn't explicitly use throw, since it calls something that says it throws MyException

# Catching Exceptions

Exceptions can be caught with `try...catch`, stopping them from moving up

# Catching Exceptions

## Exceptions can be caught with `try...catch`, stopping them from moving up

```
try {
   myMethod();
} catch (MyException e) {
   System.out.println(e.toString());
}
System.out.println("GETS HERE");
```

# Example:

# `CatchException.java`