Chapter 6
Making Decisions

Conditional execution of statements
Simple boolean expressions

- **Boolean literal**
  - one of the values `true` or `false`

- **Boolean variable**
  - A variable of type boolean. It can have one of the values `true` or `false`

- **Boolean expression**
  - expressions that evaluates to one of the values `true` or `false`
### Comparisons operators (1)

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Mathematical Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>≥</td>
<td>greater or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>≤</td>
<td>less or equal</td>
</tr>
<tr>
<td>==</td>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>≠</td>
<td>not equal</td>
</tr>
</tbody>
</table>
Comparison operators (2)

Relational Expressions ( \( >, \geq, <, \leq \) )

\[ \text{ArithmeticExpression1 \ RelationalOperator \ ArithmeticExpression2} \]

Equality Expressions ( \( ==, \neq \) )

\[ \text{Expression1 \ EqualityOperator \ Expression2} \]
Simple boolean expressions

- month == 3
  - true only if month has the value 3
- k % n == 0
  - true only if k is divisible by n
- b*b - 4.0 * a * c >= 0
  - true only if the quadratic equation $ax^2 + bx + c = 0$ has real roots
Pseudo-code if-else

\[
\text{IF } \quad \text{BooleanExpression} \quad \text{THEN} \\
\quad \text{Statements A} \\
\text{ELSE} \\
\quad \text{Statements B} \\
\text{END IF}
\]
Java template for if-else

```java
if (BooleanExpression) {
    Statements A
} else {
    Statements B
}
```

Execute if `BooleanExpression` is true

Execute if `BooleanExpression` is false
Flow chart for if-else

Boolean Expression

true

false

Statements A

Statements B
Example: absolute value

double abs(double x)
{
    if (x >= 0)
    {
        return x;
    }
    else
    {
        return -x;
    }
}

The abs function is also available in the Math library.
private boolean realRoots;
...
public void doCalculations()
{
    double d1 = b*b - 4*a*c;
    if (d1 >= 0)
    {
        double d = Math.sqrt(d1);
        root1 = (-b - d) / (2.0 * a);
        root2 = (-b + d) / (2.0 * a);
        realRoots = true;
    }
    else realRoots = false;
}

public boolean hasRealRoots()
{
    return realRoots;
}

See BlueJ project chapter6/root-finder
Block declaration of variables

- Block: a sequence of statements delimited by braces {...}
- Variables are defined in blocks and are said to have block scope
- They are undefined outside the block in which they are declared.
- We have seen three kinds of blocks.
Kinds of blocks

- Data fields have widest scope and are defined anywhere in the class
  - Example: \(a, b, c\) in QuadraticRootFinder

- Local variables in constructor or method
  - Example: \(d1\), in the \texttt{doCalculations} method

- Local variables declared in an if or else block
  - Example: \(d\), in the if-block of the \texttt{doCalculations} method
If statement with no else

IF BooleanExpression THEN
  Statements
END IF
Flow chart for if with no else

- **Boolean Expression**
- **true**
- **false**
- **Statements**
A variation of doCalculations

```java
public void doCalculations()
{
    realRoots = false;
    double d1 = b*b - 4*a*c;
    if (d1 >= 0)
    {
        double d = Math.sqrt(d1);
        root1 = (-b - d) / (2.0 * a);
        root2 = (-b + d) / (2.0 * a);
        realRoots = true;
    }
}
```

This version uses an if with no else
public void doCalculations()
{
    double d1 = b*b - 4*a*c;
    realRoots = d1 >= 0;
    if (realRoots)
    {
        double d = Math.sqrt(d1);
        root1 = (-b - d) / (2.0 * a);
        root2 = (-b + d) / (2.0 * a);
    }
}
The one-line if statement

```c
int max = x;
if (y > max) {
    max = y;
}
```

```c
int max = x;
if (y > max) max = y;
```

compute the maximum of the given variables x, y

parentheses are not needed if there is only one statement in an if or else block

The one-line if statement
Floating point comparison

- Because of truncation and round-off errors, floating point numbers should not be compared for equality or inequality using the == and != operators.

- For example, 1.0 and 0.99999999 will never be equal even though we may want to consider in practice that these numbers are close enough to be considered equal.
public class FloatingPointTester1 {
    public void doTest() {
        double x = Math.pow(Math.PI, 5.0);
        if (x == y)
            System.out.println("equal");
        else
            System.out.println("not equal");
    }
}

The results are 306.0196847852814 and 306.01968478528136 so the result printed is "not equal"
Absolute error

Absolute error = |x-y| can be used as a measure of equality:

```java
if (Math.abs(x, y) <= 1E-10)
{
    System.out.println("equal");
}
else
{
    System.out.println("not equal");
}
```

See FloatingPointTester2
**Relative error**

Relative error = \(\frac{(x-y)}{x}\) can be used as a measure of equality:

```java
double relativeError = (x - y) / x;
if (Math.abs(relativeError) <= 1E-10)
{
    System.out.println("equal");
}
else
{
    System.out.println("not equal");
}
```

See FloatingPointTester3
Conditional operator (1) ?:

- Special kind of if-statement which is sometimes convenient.

```java
booleanExpression ? expressionA : expressionB
```

- It's a ternary operator (3 operands)
Conditional operator (2) ?: 

Example: Assuming integer valued expressions we can write

```c
int v = booleanExpression ? expressionA : expressionB
```

which is equivalent to the if-statement

```c
int v;
if ( booleanExpression )
    v = expressionA;
else
    v = expressionB;
```
Conditional operator examples

double abs(double x)
{
    return (x >= 0) ? x : -x;
}

int max = (x >= y) ? x : y;

double cubeRoot(double x)
{
    return (x >= 0) ? Math.pow(x, 1.0/3.0) : -Math.pow(-x, 1.0/3.0);
}
Nested if statements

double tax;
if (a >= 10000)
{
    if (a < 100000)
    {
        tax = 0.10 * a;
    }
    else // a >= 100000
    {
        tax = 0.15 * a;
    }
} 
else // a < 10000
{
    tax = 0.05 * a;
}

Tax on an amount of money a:
5% if 0 <= a < 10,000,
10% if 10,000 <= a < 100,000
15% if a >= 100,000

tax later
Multiple if statements

- The if-else statement is used in a 2-way decision process
- It can be generalized to provide a multi-way decision process which lets you efficiently decide among several different possibilities.
- \texttt{if - else if - else if ... else}
Template for if-else if

```java
if (BooleanExpression1)
{
    Statements 1
}
else if (BooleanExpression2)
{
    Statements 2
}
........
else
{
    Default statements
}
```
Flowchart for if-else if

1. expr 1
   - true: block 1
   - false: expr 2
2. expr 2
   - true: block 2
   - false: expr N
3. expr N
   - true: block N
   - false: default block
Letter grade example

```java
String letterGrade;
if (mark < 0)
    letterGrade = "";
else if (mark > 100)
    letterGrade = "";
else if (mark >= 80)
    letterGrade = "A";
else if (mark >= 70)
    letterGrade = "B";
else if (mark >= 60)
    letterGrade = "C"
else if (mark >= 50);
    letterGrade = "D"
else
    letterGrade = "F";
```

order is important

better order independent way later
Forgetting braces

```c
if (x >= y)
{
    max = x;
    min = y;
}
else
    max = y;
    min = x;
```

This is a logical (run-time) error not a syntax error

The intent here is to compute the minimum and the maximum of x and y. Since there are no braces in the else block only the statement `max = y` is part of the else block. The statement `min = x` is not part of the if-else statement and will always be executed.
Else without if error

```c
if (x >= y)
  max = x;
  min = y;
else
{
  max = y;
  min = x;
}
```

else without if

This is a syntax error

Here the braces in the if block have been omitted

```
if (x >= y)
  max = x;
```

compiler assumes this is a complete if-statement and gives an else without if error message
Dangling else (1)

This is a logical error

Compiler interpretation

Therefore nothing is displayed for marks less than 50 and Fail is displayed for marks greater than 100
Dangling else (2)

```java
if (mark >= 50) {
    if (mark <= 100)
        System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Correct version
Compound boolean expression

- Simplest boolean expression is a boolean variable.
- Simple boolean expressions can also be obtained using the 6 comparison operators.
- Compound boolean expressions are obtained by connecting simple ones together using the logical AND, OR, and NOT operators.
# Three basic logical operators

<table>
<thead>
<tr>
<th>Java Notation</th>
<th>Mathematical Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>∧</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>~ ¬</td>
<td>logical NOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(negation)</td>
</tr>
</tbody>
</table>
Compound expressions

- "Anding" $n$ boolean expressions
  
  $$b_1 \land b_2 \land \cdots \land b_n$$

- "Oring" $n$ boolean expressions
  
  $$b_1 \lor b_2 \lor \cdots \lor b_n$$
### Truth tables: AND, OR, NOT

**AND**

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \land q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**OR**

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \lor q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**NOT**

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\sim p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

**Negation**

$\sim p$

$\overline{p}$
Java operator precedence (1)

1. ( ) have the highest precedence
2. Unary -, +, and ! have equal precedence (right associative)
3. binary *, /, and % have equal precedence (left associative)
4. binary +, - have equal precedence (left associative)
Java operator precedence (2)

5 The relational operators <, <=, <, and >= have equal precedence (not associative)

6 The equality operators == and != have equal precedence (left associative)

7 binary logical && (AND) is left associative

8 binary logical || (OR) is left associative
Example

In the expression

\[ \text{mark} \geq 0 \land \text{mark} \leq 100 \]

the expressions \( \text{mark} \geq 0 \) is evaluated first. If it is false the entire expression is false. If it is true then \( \text{mark} \leq 100 \) is evaluated and finally \( \land \) is applied
Mixed boolean expressions

- Do AND first
  
  \[(a \land b) \lor c\]

- Do OR first

  \[a \land (b \lor c)\]

- The results are not the same
**Truth table for** $a \land (b \lor c)$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$b \lor c$</th>
<th>$a \land (b \lor c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

$(a \land b) \lor c$ is not the same as $a \land (b \lor c)$
Truth table for \((a \land b) \lor c\)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>(a \land b)</th>
<th>((a \land b) \lor c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

\[
\therefore \ (a \land b) \lor c \text{ is not the same as } a \land (b \lor c)
\]
DeMorgan's laws

\( \sim (a \land b) = (\sim a) \lor (\sim b) \)
\( \sim (a \lor b) = (\sim a) \land (\sim b) \)

Generalization to \( n \) variables

\( \sim (b_1 \land b_2 \land \cdots \land b_n) = (\sim b_1) \lor (\sim b_2) \lor \cdots \lor (\sim b_n) \)
\( \sim (b_1 \lor b_2 \lor \cdots \lor b_n) = (\sim b_1) \land (\sim b_2) \land \cdots \land (\sim b_n) \)
The mathematical expression $a \leq n \leq b$ means that $n$ is between $a$ and $b$.

It cannot be translated to Java as

\[ a \leq n \leq b \]

Instead it must be translated as

\[ a \leq n \land n \leq b \]
Letter grades (better version)

```java
if (80 <= mark && mark <= 100)
    letterGrade = "A";
else if (70 <= mark && mark < 80)
    letterGrade = "B";
else if (60 <= mark && mark < 70)
    letterGrade = "C";
else if (50 <= mark && mark < 60)
    letterGrade = "D";
else if (0 <= mark && mark < 50)
    letterGrade = "F";
else
    letterGrade = ""; // invalid
```

Now the order of the conditions is not important
double tax;
if (0 <= a && a < 10000)
{
    tax = 0.05 * a;
}
else if (10000 <= a && a <= 100000)
{
    tax = 0.10 * a;
}
else // a >= 100000
{
    tax = 0.15 * a;
}
The leap year problem (1)

- There are two statements that define when a year is a leap year:
  \[ S_1 = \text{the year is divisible by 4 but not by 100} \]
  - (Examples: 1988 or 1992)
  \[ S_2 = \text{the year is divisible by 400} \]
  - (Examples: 1600 or 2000)

- Many programmers didn't know about \( S_2 \) so some programs broke on Feb 29, 2000
The leap year problem (2)

- The leap year condition is
  \[ s_1 \lor s_2 \]
- Translate to logic using
  \[ a = \text{year is divisible by 4} \]
  \[ b = \text{year is not divisible by 100} \]
  \[ c = \text{year is divisible by 400} \]
- Therefore the leap year condition is
  \[ s = s_1 \lor s_2 = (a \land b) \lor c \]
The leap year problem (3)

ALGORITHM LeapYear(year)

\( a \leftarrow \) year is divisible by 4
\( b \leftarrow \) year is not divisible by 100
\( c \leftarrow \) year is divisible by 400

```plaintext
\[(a \land b) \lor c\]
```

IF \((a \land b) \lor c\) THEN
  RETURN true (year is a leap year)
ELSE
  RETURN false (year is not a leap year)
END IF
The leap year problem (4)

- Translation to Java gives the compound boolean expression
  \[(\text{year} \mod 4 == 0) \&\& (\text{year} \mod 100 != 0) || (\text{year} \mod 400 == 0)\]

- Note that \&\& is be done first because it has a higher precedence than ||. Also == and != have lower precedence than \mod
Short circuit evaluation

- In Java the expression \( p \land \land q \) is evaluated using what is called short-circuit evaluation.

- This means that \( p \) is evaluated and if it is false then \( q \) is never evaluated since \( p \land \land q \) is false for any value of \( q \).

- Similarly, in the expression \( p \lor \lor q \), \( p \) is evaluated and if it is true then \( q \) is never evaluated since \( p \lor \lor q \) is true for any value of \( q \).
Example

- If \( x \) and \( y \) are integer variables the if-statement

```java
if (x != 0 && y/x > 2)
{
  ..
}
```

does not result in division by zero when \( x \) is zero
String comparison (1)

- The char data type for characters has an order defined on it based on the 16 bit unicode value of the character ('A' < 'B' since the code for 'A' is 65 and the code for 'B' is 66 and 65 < 66)

- Strings can be compared character by character using the standard lexicographical order based on the order of the characters
String comparison (2)

- We cannot use the four relational operators `<`, `<=`, `>`, `>=`, to compare strings as we did for numbers.
- The `==` and `!=` operators can be used but they do not compare the string objects.
- Instead they compare the string references which is normally not what we want.
Incorrect use of ==

Following code shows that == compares string references not the strings themselves

```java
String s1 = "Hello";
String s2 = "Hello";
if (s1 == s2)
    System.out.println("equal");
else
    System.out.println("not-equal");
```

Here not-equal is displayed indicating that the references s1 and s2 are not the same, even though the string objects referenced are the same.
The equals method

- The String class has an equals method with prototype
  - `public boolean equals(Object obj)`
- This method compares the string objects themselves, not the references
- Compare `s1` and `s2` with `s1.equals(s2)`
- The comparison is case-sensitive so "Hello" is different than "hello".
Correct use of equals

The following code shows that equals compares the string objects

```java
String s1 = "Hello";
String s2 = "Hello";
if (s1.equals(s2))
    System.out.println("equal");
else
    System.out.println("not-equal");
```

Here equal is displayed indicating that the objects themselves are being compared, not the references.

To test for inequality use

```java
! s1.equals(s2)
```
Lexicographical ordering

- The lexicographical order of strings is based on the order of the 16-bit character codes for the characters in the String.
- The English language uses the ASCII code for punctuation, digits '0' to '9', upper case letters 'A' to 'Z' and lower case letters 'a' to 'z'.
- This 128 character set is associated with the codes 0 to 127 (subset of Unicode).
public class CharacterDecoder
{
    public int code(char c)
    {
        int code = (int) c;
        return code;
    }
}

Here the (int) typecast converts the character to its integer code

Try it with BlueJ using characters such as 'A', 'B'

project book-projects/chapter6/strings
<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-48</td>
<td>! &quot; # $ % &amp; ' ( ) * + , - . /</td>
</tr>
<tr>
<td>48-64</td>
<td>0 1 2 3 4 5 6 7 8 9 : ; &lt; = &gt; ?</td>
</tr>
<tr>
<td>64-80</td>
<td>@ A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>80-96</td>
<td>P Q R S T U V W X Y Z [ \ ] ^ _</td>
</tr>
<tr>
<td>96-112</td>
<td>` a b c d e f g h i j k l m n o</td>
</tr>
<tr>
<td>112-15</td>
<td>p q r s t u v w x y z {</td>
</tr>
</tbody>
</table>
Ordering strings

Start comparing the two strings one character at a time until either of the following conditions are true:

1. The character in one string is different from the corresponding one in the other string.
2. One of the strings ends before the other.

Example: "Frank" precedes "Fred".
Example: "Bob" precedes "Bobby".
The compareTo method

```java
public int compareTo(String s)
```

<table>
<thead>
<tr>
<th>Boolean expression</th>
<th>Meaning if true</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1.compareTo(s2) &lt; 0</td>
<td>s1 precedes s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) &lt;= 0</td>
<td>s1 precedes or is equal to s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) &gt; 0</td>
<td>s1 follows s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) &gt;= 0</td>
<td>s1 follows or is equal to s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) == 0</td>
<td>same as s1.equals(s2)</td>
</tr>
<tr>
<td>s1.compareTo(s2) != 0</td>
<td>same as ! s1.equals(s2)</td>
</tr>
</tbody>
</table>
public class StringComparer
{
    public String compare(String s1, String s2)
    {
        int result = s1.compareTo(s2);
        if (result < 0)
            return s1 + " precedes " + s2;
        else if (result == 0)
            return s1 + " equals " + s2;
        else
            return s1 + " follows " + s2;
    }
}

project book-projects/chapter6/strings
BeanShell compare

- Using BeanShell editor type in the compare method of StringComparer and evaluate it in the workspace and try statements

```java
bsh % show();
<true>
bsh % compare("one", "two");
<one precedes two>
bsh % compare("two", "one");
<two follows one>
bsh %
```

From File menu make sure you choose Capture System
Case insensitive comparison

- In a case-insensitive comparison the strings "Hello" and "hello" would be equal
- To do this use `compareToIgnoreCase` with prototype
  - `public int compareToIgnoreCase(String s)`
Methods that return boolean values are called boolean valued methods:

String class examples:

- public boolean equals(Object obj)
- public boolean equalsIgnoreCase(String s)
- public boolean startsWith(String prefix)
- public boolean endsWith(String suffix)
public boolean isLeapYear(int y)
{
    return (y % 4 == 0) && (y % 100 != 0)
        || (y % 400 == 0);
}

bsh % show();
<true>
bsh % boolean isLeapYear(int y)
{ return (y % 4 == 0) && (y % 100 != 0) ||
(y % 400==0); }
bsh % isLeapYear(2000); 
<true>
bsh % isLeapYear(2003); 
<false>
Error checking by user

- If-statement can be used to check for illegal values:

- Example: BankAccount doesn't check if the balance is sufficient for a withdrawal. User could do it as follows

```java
BankAccount myAccount = new BankAccount(...);
...
if (myAccount.getBalance() >= 5000)
{
    myAccount.withdraw(5000);
}
```
Data Encapsulation

- It is never a good idea to leave error checking up to the user.

- Ensure **data encapsulation**
  - Make all data fields private and check in all mutator methods that they are not given illegal values.
  - Check validity of all constructor arguments.
  - Do not return references to private data fields.
Reporting errors

- In a constructor
  - display error message and exit
  - throw an exception (discussed later)

- In a method
  - display error message and exit
  - throw an exception (discussed later)
  - return a boolean value or other error indicator that can be checked by the user
  - do nothing and exit the method
BankAccount withdraw (1)

- Original version

```java
public void withdraw(double amount)
{
    balance = balance - amount;
}
```

- Error checking version

```java
public boolean withdraw(double amount)
{
    boolean amountValid = (0 <= amount) &&
                          (amount <= balance);
    if (amountValid)
        balance = balance - amount;
    return amountValid;
}
```
User can check the return value

```java
BankAccount myAccount = new BankAccount(...);
...
boolean ok = myAccount.withdraw(5000);
if (! ok)
{
    // report error here and ask for a new
    // amount or cancel withdrawal
}
```

Now if the user forgets to check the balance, data integrity is still maintained.
BankAccount deposit

- Original version

```java
public void deposit(double amount)
{
    balance = balance + amount;
}
```

- Error checking version

```java
public boolean deposit(double amount)
{
    boolean amountValid = amount >= 0
    if (amountValid)
        balance = balance + amount;
    return amountValid;
}
```
Error processing with exceptions

- An **exception** is an object from an exception class containing error information.
- To **throw an exception** means to suspend execution and signal the caller of the constructor or the method, in which the error occurred, that an error condition exists.
- **Catching an exception** means to process the error that occurred.
- **Syntax:** `throw exceptionObject;`
public BankAccount(int accountNumber, 
    String ownerName, double initialBalance) 
{
    if (accountNumber <= 0)
        throw new IllegalArgumentException(
            "Account number must be positive");
    if (ownerName.equals("") || ownerName == null)
        throw new IllegalArgumentException(
            "Owner name not defined");
    if (initialBalance <= 0)
        throw new IllegalArgumentException(
            "Balance must be non-negative");
    number = accountNumber;
    name = ownerName;
    balance = initialBalance;
}
public void deposit(double amount)
{
    if (amount < 0)
        throw new IllegalArgumentException(
            "Invalid amount for deposit");
    balance = balance + amount;
}

public void withdraw(double amount)
{
    if (amount < 0 || amount > balance)
        throw new IllegalArgumentException(
            "Invalid amount for withdraw");
    balance = balance - amount;
}
BankAccount example (3)

- See project
  - book-projects/chapter6/bank-account
public class ExceptionTester
{
    public void doTest()
    {
        BankAccount b =
        new BankAccount(123, "Fred", 100);
        b.withdraw(200);
    }

    public static void main(String[] args)
    {
        new ExceptionTester().doTest();
    }
}
ExceptionTester (2)

- Command line output when exception is thrown

```
C:\book-projects\chapter6\bank-account>java ExceptionTester
Exception in thread "main"
java.lang.IllegalArgumentException: Invalid amount for withdraw
    at BankAccount.withdraw(BankAccount.java:57)
    at ExceptionTester.doTest(ExceptionTester.java:9)
    at ExceptionTester.main(ExceptionTester.java:14)
```

C:\book-projects\chapter6\bank-account>
Catching an exception

To catch and process exceptions ourselves a try-catch statement is used

```java
try {
    // statements to try that could throw
    // an exception
}
catch (... ) {
    // statements to execute when an exception
    // occurs in try block
}
// other statements to execute if no exception
// occurs
```
public class ExceptionTester
{
  public void doTest()
  {
    try
    {
      BankAccount b =
        new BankAccount(123, "Fred", 100);
      b.withdraw(200);
    }
    catch (IllegalArgumentException e)
    {
      System.out.println(e.getMessage());
    }
  }
}
The PSR game (1)

- Paper-Scissors-Rock game
- There are two players: Player 1 and Player 2
- Each player has a piece of paper (P), a pair of scissors (S), and a rock (R). Each player presents one of these objects and winner is determined using the rules.
  - P wins over R since paper covers rock
  - S wins over P since scissors cut paper
  - R wins over S since rock breaks scissors
**PSR Game decision table**

<table>
<thead>
<tr>
<th>Player 1</th>
<th>Paper</th>
<th>Player 2</th>
<th>Rock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper</td>
<td>Draw</td>
<td>Player 2</td>
<td>Player 1</td>
</tr>
<tr>
<td>Scissors</td>
<td>Player 1</td>
<td>Draw</td>
<td>Player 2</td>
</tr>
<tr>
<td>Rock</td>
<td>Player 2</td>
<td>Player 1</td>
<td>Draw</td>
</tr>
</tbody>
</table>
public class PSRPlayer
{
    private char choice;
    public PSRPlayer() {...}
    public char getChoice() {...}
    public void setChoice(char choice) {...}
}

Here choice is one of the three characters 'P' for paper, 'S' for scissors, and 'R' for rock.
public class PSRGame
{
    private PSRPlayer p1, p2;

    public static final int DRAW = 0;
    public static final int WIN_PLAYER_ONE = 1;
    public static final int WIN_PLAYER_TWO = 2;

    public PSRGame(PSRPlayer p1, PSRPlayer p2) {...}
    public int playRound() {...}
}

Here `playRound` returns one of the values `DRAW`, `WIN_PLAYER_ONE`, `WIN_PLAYER_TWO`
public class PSRPlayer
{
    private char choice;
    public PSRPlayer()
    {
    }
    public char getChoice()
    {
        return choice;
    }
    public void setChoice(char choice)
    {
        this.choice = Character.toUpperCase(choice);
    }
}
public class PSRGame
{
    private PSRPlayer p1, p2;

    public static final int DRAW = 0;
    public static final int WIN_PLAYER_ONE = 1;
    public static final int WIN_PLAYER_TWO = 2;

    public PSRGame(PSRPlayer p1, PSRPlayer p2)
    {
        this.p1 = p1;
        this.p2 = p2;
    }

    public int playRound() {...} // next slide
public int playRound()
{
    // throw exceptions here for invalid choices
    if (isWin(p1, p2)) // p1 wins over p2
    {
        return WIN_PLAYER_ONE;
    }
    else if (isWin(p2, p1)) // p2 wins over p1
    {
        return WIN_PLAYER_TWO;
    }
    else
    {
        return DRAW;
    }
}
PSRGame implementation (3)

Checking for illegal input

```java
if ( isInvalidChoice(p1) )
{
    throw new IllegalStateException(
        "Player 1: invalid choice" preferred
    );
}
if ( isInvalidChoice(p2) )
{
    throw new IllegalStateException(
        "Player 2: invalid choice"
    );
}
```
private static boolean isInvalidChoice(PSRPlayer p) {
    char choice = p.getChoice();
    return choice != 'P' && choice != 'S' &&
            choice != 'R';
}

Recall that a static method is not associated with any object of the class but an instance method is associated with an object of the class.

This method can be declared either as a static method or an instance method since it doesn't depend on the choice data field of the class.
private static boolean isWin(PSRPlayer p1, PSRPlayer p2) {
    char pc1 = p1.getChoice();
    char pc2 = p2.getChoice();
    return (pc1 == 'P' && pc2 == 'R') ||
            (pc1 == 'S' && pc2 == 'P') ||
            (pc1 == 'R' && pc2 == 'S');
}

This method can be declared either as a static method or an instance method since it doesn't depend on the choice data field of the class.
Test it with BeanShell

```bash
bsh % addClassPath("c:/book-projects/chapter6/psr-game");
bsh % PSRPlayer p1 = new PSRPlayer();
bsh % PSRPlayer p2 = new PSRPlayer();
bsh % PSRGame game = new PSRGame(p1, p2);
bsh % p1.setChoice('p'); p2.setChoice('s');
bsh % print(game.playRound());
2
bsh % p1.setChoice('s'); p2.setChoice('p');
bsh % print(game.playRound());
1
bsh % p1.setChoice('p'); p2.setChoice('p');
bsh % print(game.playRound());
0
bsh %
```
Test it with BlueJ

- Create a PSRPlayer object for player 1
- Create a PSRPlayer object for player 2
- Create a PSRGame object of these players
- Use setChoice method for each player
- Use playRound method to see outcome of game
- repeat last two steps
Command line interface

We can write a class called PSRGameRunner that can run get keyboard input from the user and run the game from the terminal window (command prompt window):

Player 1, enter your choice: P, S, or R
P
Player 2, enter your choice: P, S, or R
s
Player 2 wins
The Scanner class in Java 5 can be used to get input from the console. Import it using

```
import java.util.Scanner;
```

Construct a Scanner object using

```
Scanner input = new Scanner(System.in);
```

Use the nextInt method to read the next number typed in the console window and return it as an int value. Prototype is

```
public int nextInt()
```
Use the `nextLong` method to read the next number typed in the console window and return it as a long value. Prototype is

```java
public long nextLong()
```

Use the `nextFloat` method to read the next number typed in the console window and return it as a float value. Prototype is

```java
public float nextFloat()
```

Use the `nextDouble` method to read the next number typed in the console window and return it as a double value. Prototype is

```java
public double nextDouble()
```
Use the nextLine method to read the rest of the line typed in the console window and return it as a String. The end of line character is read and thrown away.

```java
public String nextLine()
```
If you are not careful there are some pitfalls when using the Scanner class for interactive input. For example, the statements

```java
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
System.out.println("Enter your name");
String name = input.nextLine();
```

The result will be that age is assigned correctly but the empty string will be assigned to name because when you entered age and pressed the Enter key the end of line was left in the input buffer and was read as name by the nextLine statement.
### Scanner input with Java 5 (4)

The following version corrects this

```java
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
input.nextLine(); // eat the end of line
System.out.println("Enter your name");
String name = input.nextLine();
```

Now name is assigned the correct result since the statement

```java
input.nextLine(); // eat the end of line
```
reads the end of line in the buffer and throws it away.
import java.util.Scanner;
public class PSRGameRunner
{
    public void run() // for BlueJ
    {
        Scanner input = new Scanner(System.in);
        PSRPlayer p1 = new PSRPlayer();
        PSRPlayer p2 = new PSRPlayer();
        PSRGame game = new PSRGame(p1, p2);

        System.out.println(
            "Player 1, enter your choice: P, S, or R");
        char player1Choice = input.nextLine().charAt(0);
        p1.setChoice(player1Choice);
System.out.println("Player 2, enter your choice: P, S, or R");
char player2Choice = input.nextLine().charAt(0);
p2.setChoice(player2Choice);

int result = 0;
try {
    result = game.playRound();
} catch (IllegalStateException e) {
    System.out.println("Illegal input");
    result = PSRGame.DRAW;
}
if (result == PSRGame.WIN_PLAYER_ONE) System.out.println("Player 1 wins!");
else if (result == PSRGame.WIN_PLAYER_TWO) System.out.println("Player 2 wins!");
else System.out.println("It's a draw");

public static void main(String[] args) {
    PSRGameRunner program = new PSRGameRunner();
    program.run();
}
The main method (1)

- To run a class outside the BlueJ environment at the terminal window (command prompt) it is necessary that the class have a main method with the structure

```java
public static void main(String[] args)
{
    // create some objects
}
```

- The command line java interpreter (Java Runtime Environment (JRE)) will begin by executing any statements in this method
The main method (2)

- PSR example

```java
public static void main(String[] args) {
    PSRGameRunner program = new PSRGameRunner();
    program.run();
}
```

- In this case the JRE will create a PSRGameRunner object and call its run method.
Command line (Windows/Unix)

```
java PSRGameRunner
Player 1, enter your choice: P, S, or R
p
Player 2, enter your choice: P, S, or R
s
Player 2 wins!
```

Here we use the command line java interpreter outside BlueJ.
Complex roots of quadratic

We didn't do this in COSC 1046. See pages 294 to 399